Elevating Jupyter Notebook Maintenance Tooling by Identifying and Extracting Notebook Structures

Yuan Jiang
Carnegie Mellon University

Christian Kästner
Carnegie Mellon University

Shurui Zhou
University of Toronto

***Abstract*—Data analysis is an exploratory, interactive, and often collaborative process. Computational notebooks have become a popular tool to support this process, among others because of their ability to interleave code, narrative text, and results. However, notebooks in practice are often criticized as hard to maintain and being of low code quality, including problems such as unused or duplicated code and out-of-order code execution. Data scientists can benefit from better tool support when maintaining and evolving notebooks. We argue that central to such tool support is identifying the structure of notebooks. We present a lightweight and accurate approach to extract notebook structure and outline several ways such structure can be used to improve maintenance tooling for notebooks, including navigation and finding alternatives.**

I. Introduction

Data science is a field that extracts insights from data and applies these insights across a broad range of applications. Data science work is usually exploratory and iterative, and often collaborative [1]–[3]. *Computational notebooks* enable their users to interleave code, visualizations, and narrative texts in a single document [2]. They have become the primary coding environment for data scientists, with millions of data science notebooks shared publicly each year [4].

While computational notebooks are very popular among data scientists, many practitioners and researchers report problems [5], [6]. Previous work examining millions of notebooks and dozens of interviews has shown that many notebooks are "messy" and most contain minimal to no documentation and structuring (in markdown cells) that could facilitate easy understanding [5], [7]. Understanding is essential for collaboration, reuse, and maintenance though. Poor quality code in public notebooks makes them unreliable for inexperienced learners [8]. Common problems manifest in dead-ends, duplicated code, and tangled or scattered code [9]. Our goal is to make it easier to build tooling that helps notebook practitioners understand, navigate, modularize, and maintain notebook code.

In this paper, we lay the foundation for maintenance tooling with an efficient algorithm to identify and extract Jupyter notebook structures as labeled dependency graphs, as summarized in Figure 1. We automatically label each notebook cell with machine learning (ML) stages (e.g., data collection, training, evaluation) and extract data dependency relations among the cells. Each labeled node in the output graph represents a code cell, and every directed edge represents a def-use relation between a pair of cells. In a preliminary evaluation, we show that our approach is accurate and very lightweight, outperforming prior approaches in terms of lower complexity, higher accuracy, and lower execution time. Finally, we discuss potential tooling based on our labeled dependency graph by sketching a navigation tool and reporting structural patterns commonly found in notebooks.

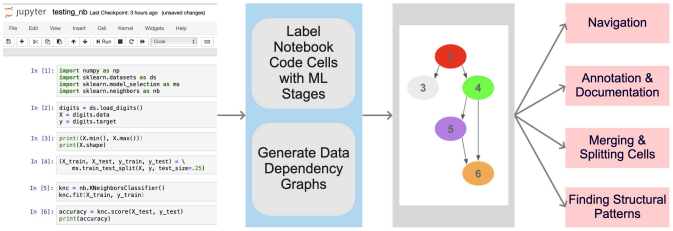

Fig. 1. Summary of Our Workflow: a Jupyter notebook is passed to our algorithm, which labels notebook cells with ML stages and generates a data dependency graph. The labeled dependency graph can be useful for applications such as navigation, annotation and documentation generation, merging and splitting cells, and finding structural patterns in a notebook dataset.

We make all implementation code and manually labeled data available at github.com/cindyyuanjiang/Jupyter-Notebook-Project.

II. Background & Related Work

A computational notebook is an interactive literate programming document which is executed in the computational environment; Python notebooks in the Jupyter environment are the most popular of these [7]. Literate programming refers to the concept of combining code and natural language which allows programmers to express their thoughts behind the logic of a program [3]. An interactive computational notebook environment allows code parts, known as cells, to be executed incrementally to produce immediate results and visualizations. Because notebook users are free to execute any cell at any time, the execution marks may not be in chronological order from top down.

A. Previous Analysis or Tools to Improve Jupyter Notebooks

Coding practices in notebooks and the popular computational notebook environments like Jupyter themselves have been studied extensively (e.g., [3], [5], [7], [9], [10]), revealing many poor practices and pain points that hamper understanding and maintenance. Many researchers have subsequently tried to address various problems through improved tooling.

A common theme are attempts to improve documentation: Wang et al. [9] implemented a deep-learning-based automated documentation generation system, creating documentation for source code, retrieving online API documentation for source code, and nudging users to write documentation. Yang et al. [11] used program synthesis techniques and dynamic program analysis to generate documentation for data wrangling code which summarizes data transformations on representative examples from data. Rule et al. [12] designed a Jupyter notebook extension for cell folding to aid navigation and comprehension.

Other tooling focuses on managing variants and revisions: For example, Kery et al. [13] designed and implemented a lightweight local versioning plugin into Jupyter notebooks for data scientists to better explore and understand their past analysis choices, using algorithmic and visualization techniques. Head et al. [6] introduced code gathering tools to help data scientists clean and recover different versions of code in cluttered notebooks using software slicing.

Finally, several papers focus on supporting data scientists with structuring their code into ordered cells: Titov et al. [14] proposed an algorithm for automatically resplitting cells into more semantically cohesive units. Wenskovitch et al. [15] designed a visualization approach to communication by displaying the dependencies between the cells of a notebook, using dynamic analysis.

Each of these tools developed custom infrastructure from scratch. We aim to encourage more maintenance tooling for notebooks by providing a common underlying analysis infrastructure that can extract the structure in a notebook.

B. Labeling Notebook Cells

In addition to dependencies between cells, it is often useful to understand what different parts of a notebook are doing. Data science code is often structured according to a conceptual *data science pipeline*, which starting from model requirements, considers data collection, data cleaning, labeling, feature engineering, model training, model evaluation, and deployment [16]. In notebooks, particularly data cleaning and feature engineering (collectively called data wrangling [11], [17]), model training, and model evaluation are common.

Understanding which pipeline stages correspond to which notebook cells can be helpful for various understanding and maintenance tasks and is a core of our approach. Past approaches to identify stages either relied on very simple heuristics or relied heavily on expensive ML classification. On one end, Venkatesh et al. [18] simply labeled cells by API calls contained in them; on the other end, Zhang et al. [17] used a weakly supervised transformer architecture to classify code snippets which jointly models data science code and natural language annotations. Our proposed work outperforms both of these approaches, providing labels accurately and fast.

III. Methods

To provide the foundation for more maintenance tooling for notebooks, to address their common "messy" and undocumented nature in an exploratory and iterative workflow, we develop an algorithm to identify and extract structures from Jupyter notebooks as directed, labeled dependency graphs where every node represents a code block (usually a notebook cell), every edge represents a data dependency relation, and nodes are labeled corresponding to their stages in the ML pipeline. In Figure 3, we illustrate the resulting output graph for an excerpt of a notebook. By default, we use notebook cells as the granularity for graph nodes because an ideal notebook cell can be viewed as a proto-function and reused to do one dedicated action [14]. To build the labeled dependency graph, we proceed in two steps: identifying dependencies between cells and mapping cells to ML stages.

A. Data Dependency

We used standard data flow analysis to identify def-use chains in a notebook's code. We then group these dependencies by code blocks (cells), representing dependencies among cells as directed edges in our graph.

We and others [7] found that most notebook code is fairly simple, hence even fairly simple and fast data-flow analysis provides accurate results (e.g., context sensitivity and pointer analysis add little benefit). We largely reused the static data-flow analysis from the *python-program-analysis* package developed by Microsoft [9] and modified it as follows: First, we did not track dependencies from import statements because they obfuscate the dependency graph without adding value for maintenance tasks. Second, we made the tool conservative with regards to dependencies resulting from function side effects, assuming that a function call might modify its arguments, therefore treating the function as a definition site of its arguments. We made the latter change in preferring occasional false positive dependencies between cells over missing edges or high analysis costs from inter-procedural analysis of library code.

B. Identifying ML Stages

We label each node in our dependency graph with a corresponding stage of the ML pipeline. Commonly, an ML pipeline consists of data collection, data cleaning, labeling, feature engineering, model training, and model evaluation [16]. As in prior work [11], [17], we combine data cleaning, labeling, and feature engineering collectively as data wrangling to avoid potential overlapping of their meanings. We define the most relevant stages – *Data Collection*, *Data Wrangling*, *Training*, *Evaluation*, and *Exploration* – with corresponding examples in Figure 2. Our labels are similar to prior classification by Zhang et al. [17] because they are all based on standard stages, but we do not include the *Import* stage because it is not very important from a maintenance perspective.

Human developers can map most cells clearly to one or multiple of these stages (as we will show in our evaluation). While investigating notebooks, we found that some cells may correspond to multiple stages, for example, both perform feature engineering and data exploration in the same cell. Usually though one stage is clearly the dominant purpose of a cell. To avoid the complexity of having multiple labels, we agreed on

Stage Name	Definition/Description	Example
Data Collection	Data Collection cells load data that will often be passed to other stages in the notebook.	`import sklearn.datasets as ds` `digits = ds.load_digits()`

Data Wrangling	Data wrangling cells clean, transform, or filter data loaded from the data collection stage. These cells also perform feature engineering on the collected data.	`from sklearn.preprocessing import OneHotEncoder`
		`enc = OneHotEncoder(handle_unknown='ignore')`
		`enc.fit(data)`
		`from sklearn.decomposition import PCA`
		`pca = PCA(n_components=2)`
		`pca.fit(X)`

Training	Training cells define and fit supervised-learning ML models to the collected data to predict on some feature of the data.	`logreg = lm.LogisticRegression()` `logreg.fit(X_train, y_train)`

Evaluation	Evaluation cells predict a feature of some dataset using ML model(s) or measure/compute the accuracy or explanatory power of the model(s).	`y_predicted = logreg.predict(X_test)`
		`import sklearn.model_selection as ms`
		`ms.cross_val_score(logreg, X, y)`

Training+Evaluation	Training+Evaluation cells include both training and evaluation stages, as described above.	`logreg = lm.LogisticRegression()`
		`logreg.fit(X_train, y_train)`
		`y_predicted = logreg.predict(X_test)`
		`import sklearn.model_selection as ms`
		`ms.cross_val_score(logreg, X, y)`

Exploration	Exploration cells visualize, print or plot data or data's relevant information (e.g. shape) and plot or print results related to the training or evaluation process. Unsupervised learning is also classified as exploration.	`fig, ax = plt.subplots(1, 1, figsize=(8, 3))`
		`ax.imshow(np.vstack((y_test, y_predicted)),`
		`interpolation='none', cmap='bone')`
		`ax.set_axis_off()`
		`ax.set_title("Actual and predicted survival outcomes "`
		`"on the test set")`
		`from sklearn.cluster import KMeans`
		`import numpy as np`
		`X = np.array([[1, 2], [1, 4], [1, 0],`
		`[10, 2], [10, 4], [10, 0]])`
		`kmeans = KMeans(n_clusters=2, random_state=0).fit(X)`

Fig. 2. ML Stage Definitions

a priority order, assigning always the stage with the highest priority if multiple stages may apply. As sole exception, we introduce a dedicated label for cells that perform both training and evaluation, as they often co-occur and neither stage should be considered as subsumed by the other. Our final priority order is: *Training+Evaluation* > *Training* or *Evaluation* > *Data Collection* > *Data Wrangling* > *Exploration*. If a cell does not correspond to any stage, we label it as *"N/A"*.

While there are several different strategies to identify stages for code fragment, we develop a simple but accurate heuristics-based approach that does not rely on textual documentation in the notebook and avoids computationally expensive and brittle ML techniques.

As recognized in prior work [4], [15], data science code often uses a small set of popular libraries for typical ML activities. We use knowledge about such APIs as the seed for our labels. We build an API-to-stage mapping for commonly used ML libraries (currently *scikit-learn*, *Keras*, and *pandas*). We map API calls to specific stages by inspecting their functionalities in the respective official API references. To correctly distinguish API calls with the same name (e.g., *'fit'* used for *Training* in *KNeighborsClassifier* or used for *Data Wrangling* in *PCA* in *scikit-learn*), we use a type inference tool *pyright* [19] to identify which library class makes the API call.

We use known APIs as seeds to identify stages for a cell and propagate information from there along data-flow edges. We found that identifying a cell's stage solely by API calls contained in the cell is insufficient, but that notebook users often put logically related statements in the same cell or structurally close to one another. We hence propagate information as follows: Every time we analyze a statement, we consider two scenarios based on the number of *child statements* it has according to the data-flow analysis.

- *One child statement*: if the current statement and its child statement are in the same code cell or the current statement is the closest parent to the child statement with regard to their location in the source code, we propagate the current statement's labels to the child statement.
- *Multiple child statements*: for every pair of the current statement and one of its child statements, we follow the same mechanism in the previous case.

After the algorithm traverses every data-flow edge between statements in the source code for propagating information, it labels each cell with the highest-priority label existing in that cell.

IV. Preliminary Evaluation

To be useful in tooling for maintenance and evolution, our labeled dependency graph needs to be *accurate* and *fast to compute*. The latter is important both when analyzing many notebooks (e.g., when indexing reusable structures for search) and when computing analyses in the background (e.g., within a notebook plugin).

First, we evaluate the accuracy of our algorithm, especially cell labelings. Second, we measure the performance of each step in our algorithm.

A. Dataset

We assemble a dataset of *all* public notebooks scraped from GitHub repositories created on two specific days, January 1 (704 notebooks) and January 6 (1629 notebooks), 2021. Both are weekdays, though January 1 is a holiday in many countries. We expect that January 1 skews more toward hobbyists whereas January 6 represents a more typical workday, together covering a comprehensive representative of notebooks on GitHub. We sample by release days rather than popularity to get a full cross section of notebooks typically published on GitHub; we evaluate on recent notebooks that represent the state of practice now, rather than performing longitudinal analysis of historic data.

B. Accuracy

To evaluate the correctness of the output data dependency graphs, we need to measure the accuracy of the node labelings and the dependencies between cells.

Accuracy of Node Labelings. To evaluate accuracy, we need to establish ground truth of the correct label. We establish ground truth manually.

To assure reliability of manual labeling, we first created explicit labeling instructions and evaluated inter-rater agreement among three labelers. Specifically, two authors independently labeled 102 and 153 cells from two different sets of 6 notebooks randomly selected from our dataset, and a third author independently labeled all 255 cells of these 12 notebooks, until each cell had two independent labels. We computed agreement with Cohen's kappa and discussed disagreements

between raters. We then refined the instructions and repeated the process for 434 cells of another 11 notebooks. After the second round, we reached a kappa score of 0.83, which is generally interpreted as almost perfect agreement [20], suggesting that manual labeling is indeed reliable.

After establishing reliability, we then manually label 1208 cells from 50 notebooks as ground truth, 25 fresh notebooks randomly selected each from the January 1 and January 6 datasets. We run our algorithm on these notebooks and compare the results against experts' manual labels. Automated labels match our ground truth in 903 out of 1208 cells, for 75% accuracy (compared to 38% accuracy for a simple baseline predictor that always predicts the majority label *Data Wrangling*).

To better understand the sources of inaccuracy, we explored the confusion matrix (can be found on the project website). Almost half of the errors (44%) come from mislabeling *Exploration* and *Data Wrangling* cells as *"N/A"*. Distinguishing among *Exploration*, *Data Wrangling*, and *"N/A"* could be difficult in some scenarios. For instance, some *Data Wrangling* processes are unidentified because they do not call any *Data Wrangling* APIs nor are they near them, thus require deeper understanding of the code's context to identify the stage. Another problem is that some cells make *Data Wrangling* API calls, but only intend to explore the data. We have not yet found a way to better distinguish these cases heuristically.

We compare our accuracy result against two previously discussed approaches on identifying ML stages for notebook cells. Venkatesh et al. [18] labeled cells solely by API calls contained in them, though they did not evaluate accuracy or release their implementation. We approximate their approach by running our implementation without type inference and information propagation along data-flow edges. This results in an accuracy of 69%, showing how our improvements correspond to a 19% reduction in error over merely identifying API calls within a cell. Zhang et al. [17] used ML techniques to classify code snippets based on content and context. We were unable to reproduce their results (we only achieved 12% accuracy replicating their methods on their dataset), but the paper reports 70% accuracy for a very similar task. This indicates that our *much simpler* approach can achieve a reduction in error of 17% over their reported numbers.

Accuracy of Cell Dependencies. Establishing ground truth for cell dependencies is tedious. We opted to not perform a systematic evaluation, but instead rely on manual inspection of analysis results in the sampled notebooks. We found occasional spurious edges from conservative assumptions in our analysis, but no substantial problems.

C. Performance

We measure execution times of our analysis using a commodity laptop (2.8 GHz Quad-Core Intel Core i7, Intel Iris Plus Graphics 655 1536MB, 16GB memory) for all 2333 notebooks in our dataset and report times separately for the three steps. The slowest component is *Type Inference File Generation* using the off-the-shelf tool *pyright* [19], which is used to disambiguate API calls. All other components can be executed in much under one second for almost all notebooks, see Table I. Assuming type inference information can be cached (or improved with a different tool), the entire analysis for a full notebook can be performed in 331ms on average, fast enough to run in the background during interactive use.

TABLE I
AVERAGE RUNTIME PER NOTEBOOK FOR EACH STEP IN THE METHODS.

Step in Methods	Avg. Runtime
Type Inference File Generation	3720 ms
Seed Function Identification & Data Flow Analysis	144 ms
Information Propagation & Labeled Graph Generation	187 ms

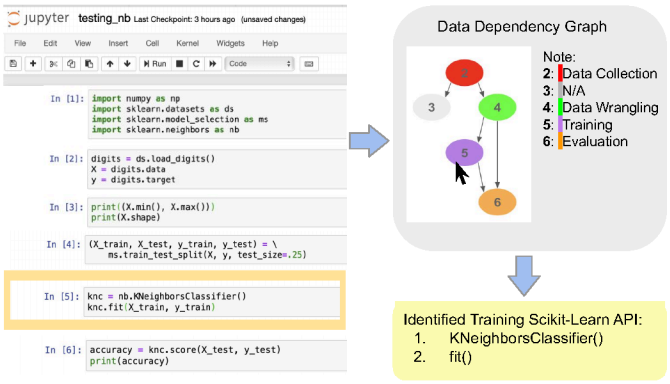

Fig. 3. Sketch of Navigation Tool Prototype

Previous work by Zhang et al. [17] used an ML architecture to predict stages for notebook cells. The paper did not report any performance numbers. While we were not able to exactly replicate their approach and did not receive access to pretrained models, we could train a smaller model using the paper's script (at significant one time training cost). Even with the smaller model, label inference took 2007ms per notebook in the provided test dataset. That is, our much simpler (and more accurate) approach seems more feasible for interactive settings.

V. Examples of Anticipated Applications

We believe the labeled dependency graph is a *useful* foundation for many tools by providing support for maintenance and evolution for notebooks specifically and data-science pipelines more generally. Here we outline examples of envisioned tooling.

Navigation. Most obviously, we expect that visualizations of the graph will be useful for navigating in a notebook along dependency edges (e.g., jumping over deadends or cells that perform exploration) or navigating directly to code of specific stages. We sketch a simple visualization in Figure 3. A plugin could link nodes in the graph with cells in the notebook in both directions. It could further highlight through which variables or cells are dependent or why cells are identified as belonging to a specific stage. Highlighting the cells helps the users to track

where the cells are and how the dependencies are reflected in the notebook.

Notebook Patterns. Extracting structures from a large set of notebooks allows us to find patterns among them, useful for a variety of tasks. Users can search over code structures of public notebooks. A plugin may highlight alternative cells to the one currently edited. Analysis tools might indicate when a user's notebook has an unusual structure. Researchers and tool builders can learn about common or uncommon patterns and use this information to develop tools that are useful for a large number of notebook users. Our graph provides a good abstraction for analyzing patterns.

As an example, we identify (1) when notebooks train multiple models in parallel (models trained independently in different cells on shared or separate input data), (2) when they compare the results of multiple models, and (3) when they contain deadends. We record the number of these pattern occurrences over all 2333 notebooks from the January 1 and January 6 datasets.

Parallel training processes happen when users explore multiple ML training models on a shared or separate datasets. In such settings, developer tools could help to prune no longer needed branches, merge branches, or even make manually explored differences accessible to AutoML tools. Among all 2333 notebooks in our dataset, 169 notebooks contain parallel training processes on a shared dataset and 575 notebooks contain parallel training processes on separate datasets. In total, 32% of the notebooks explore alternatives in training processes.

In contrast, explicit comparison between different evaluation processes is rare. We found only 83 notebooks among the ones analyzed, which accounts for less than 4% of all notebooks. It seems more common to simply print accuracy numbers and to compare them manually than to compare them in code.

Finally, deadends – data wrangling or exploration cells with no children in the data dependency graphs – occur in almost every notebook analyzed (94%). Tooling could suggest cleanup mechanisms, manual or automated.

VI. Conclusion

We implemented an efficient algorithm to identify and extract Jupyter notebook structures as labeled data dependency graphs, where nodes represent cells and directed edges represent data dependency relations among cells. The algorithm involves generating data dependency information of cells and labeling cells with ML stages. Our evaluation shows that our methods achieve high accuracy for labeling cells and fast runtime performance. We sketch a navigation tool prototype using data dependency graphs generated from our methods and discuss a number of patterns in our notebook dataset. Given the efficient runtime, tool builders can run our analysis in the browser background and use our data dependency graphs for various purposes like navigation, documentation generation, or learning about notebook structures in general.

Acknowledgment

Kästner's work was supported in part by NSF award 2131477. Zhou's work was supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN2021-03538. We thank Chenyang Yang for his help with generating our dataset.

References

[1] N. Nahar, S. Zhou, G. Lewis, and C. Kästner, "Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process," in *Proc. ICSE*, 2022.

[2] K. Patel, J. Fogarty, J. A. Landay, and B. Harrison, "Investigating statistical machine learning as a tool for software development," in *Proc. CHI*, pp. 667–676, 2008.

[3] 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 *Proc. CHI*, 2018.

[4] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer, "Data science through the looking glass and what we found there," *ArXiv*, vol. abs/1912.09536, 2019.

[5] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in *Proc. CHI*, 2020.

[6] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Proc. CHI*, 2019.

[7] J. F. Pimentel, L. G. P. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter notebooks," in *Proc. Conf. Mining Software Repositories (MSR)*, pp. 507–517, 2019.

[8] J. Wang, L. Li, and A. Zeller, "Better code, better sharing: On the need of analyzing Jupyter notebooks," in *Proc. ICSE-NIER*, p. 53–56, 2020.

[9] A. Y. Wang, D. Wang, J. Drozdal, M. J. 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 Transactions on Computer-Human Interaction*, vol. 29, 2022.

[10] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," *Proc. CHI*, 2018.

[11] C. Yang, S. Zhou, J. L. C. Guo, and C. Kästner, "Subtle bugs everywhere: Generating documentation for data wrangling code," in *Proc. ASE*, pp. 304–316, 2021.

[12] A. Rule, I. Drosos, A. Tabard, and J. D. Hollan, "Aiding collaborative reuse of computational notebooks with annotated cell folding," in *Proc. CHI*, 2018.

[13] M. B. Kery and B. A. Myers, "Interactions for untangling messy history in a computational notebook," in *Proc. Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 147–155, 2018.

[14] S. D. Titov, Y. Golubev, and T. Bryksin, "Resplit: Improving the structure of Jupyter notebooks by re-splitting their cells," *ArXiv*, vol. abs/2112.14825, 2021.

[15] J. E. Wenskovitch, J. Zhao, S. A. Carter, M. L. Cooper, and C. North, "Albireo: An interactive tool for visually summarizing computational notebook structure," in *Proc. Visualization in Data Science (VDS)*, pp. 1–10, 2019.

[16] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *Proc. ICSE-SEIP*, 2019.

[17] G. Zhang, M. Merrill, Y. Liu, J. Heer, and T. Althoff, "Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis," *EPJ Data Science*, vol. 11, 2022.

[18] A. P. S. Venkatesh and E. Bodden, "Automated cell header generator for Jupyter notebooks," in *Proc. International Workshop on AI and Software Testing/Analysis*, p. 17–20, 2021.

[19] Microsoft, "Pyright." https://github.com/microsoft/pyright, 2020.

[20] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.