Enhancing Computational Notebooks with Code+Data Space Versioning

HANXI FANG, University of Illinois at Urbana-Champaign
SUPAWIT CHOCKCHOWWAT, University of Illinois at Urbana-Champaign
HARI SUNDARAM, University of Illinois at Urbana-Champaign
YONGJOO PARK, University of Illinois at Urbana-Champaign

There is a significant gap between how people explore data and how Jupyter-like computational notebooks are designed. People explore data nonlinearly, using execution undos, branching, and/or complete reverts, whereas computational notebooks are designed for sequential exploration only. Recent works like ForkIt are still insufficient to support these multiple modes of nonlinear exploration in a unified way.

In this work, we address the challenge by proposing two-dimensional code+data space versioning for computational notebooks and verifying its effectiveness using our prototype, Kishuboard, which seamlessly integrates with Jupyter. By adjusting code and data knobs, users of Kishuboard can intuitively manage the state of computational notebooks in a flexible way, thereby achieving both execution rollbacks and checkouts across complex multi-branch exploration history. Moreover, this two-dimensional versioning mechanism can easily be presented along with a friendly one-dimensional history. Human subject studies indicate that Kishuboard can significantly enhance user productivity in various data science tasks.

ACM Reference Format:
Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2024. Enhancing Computational Notebooks with Code+Data Space Versioning. 1, 1 (October 2024), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There is a significant gap between how people explore data and how computational notebooks (e.g., Jupyter [19], Colab [12]) are designed. This discrepancy causes pain and wasted resources. People explore data *nonlinearly*, while computational notebooks are designed for *sequential* exploration only. In Fig 1, for example, Alice evaluates a model after dropping the `age` feature from the data (left). For higher accuracy, she wishes to test another feature set. *If allowed,* she could **roll back** some of the executions (left to middle), edit the feature selection (middle), and re-run the training logic (right). The new model shows lower accuracy. Alice would end this alternative exploration by **checking out** both code and data from an earlier version (right to left). Unfortunately, none of the existing off-the-shelf computational notebooks—from open-source tools (e.g., Jupyter [19], R Markdown [5]) to commercial services (e.g., Google Colab [12], Mode Analytics [3])—allow this nonlinear workflow. If they offer nonlinear exploration, it would completely reshape how data scientists explore data, while immediately addressing the pressing needs shared by practitioners (§3).

Authors' addresses: Hanxi Fang, University of Illinois at Urbana-Champaign, hanxif2@illinois.edu; Supawit Chockchowwat, University of Illinois at Urbana-Champaign, supawit2@illinois.edu; Hari Sundaram, University of Illinois at Urbana-Champaign, hs1@illinois.edu; Yongjoo Park, University of Illinois at Urbana-Champaign, yongjoo@illinois.edu.

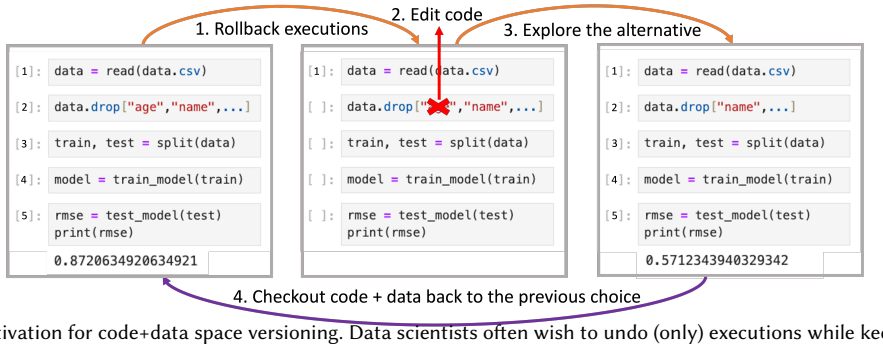
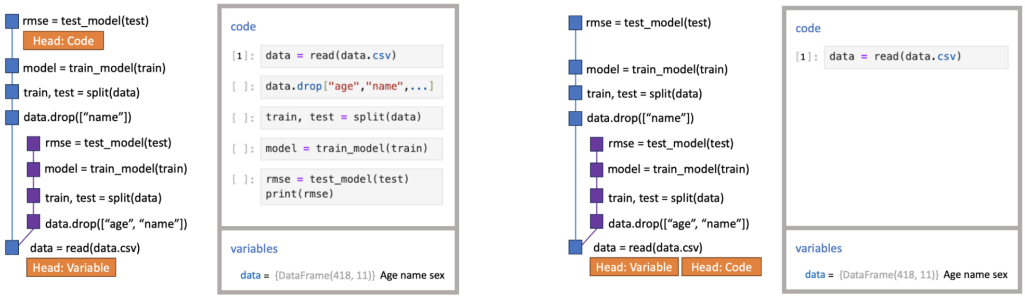


Fig. 1. The motivation for code+data space versioning. Data scientists often wish to undo (only) executions while keeping the code the same (i.e., **execution rollback**; left to middle) for testing alternative methods (e.g., edits in the middle followed by executions in the right); Or they often wish to completely revert all of their activities, jumping back to a certain point in the past (i.e., **checkout**; right to left). Our goal is to enable these various types of code/data state modifications through an intuitive user interface.

Novel interfaces and systems are proposed to narrow this gap; however, they remain insufficient. For example, ForkIt [54] allows for pursuing multiple exploration paths simultaneously with side-by-side columns; each column represents an independent path branched out from a common ancestor. While it is a significant improvement over sequential exploration, ForkIt requires *proactive* planning without the ability to *retrospectively* branch out and return. Moreover, it is unclear how users can manage more than a handful of exploration paths, each of which may recursively branch into more alternatives. Approaches like Diff-in-the-Loop [49] and Chameleon [16] offer visual aids for comparing data between iterative explorations. While these visualization techniques are helpful, they can be more successful when the underlying systems, like computational notebooks, are designed to support iterations through nonlinear exploration, which is lacking today. Recording code execution order [15, 21, 22, 31] and highlighting branched code structures [20, 40, 47, 48, 53] are also beneficial, but they cannot manipulate the data state of computational notebooks, which is required for the workflow described in Fig 1.

In this work, we propose two-dimensional code+data space versioning for computational notebooks, allowing novel primitives, **execution rollback** and **code+data checkout**, for *consistently* navigating past states. In Fig 2, for example, users can **roll back** executions by moving the "Head: Variable" box from the latest commit (i.e., `rmse=test_model(test)`



(a) Code and data states for **execution rollback** (b) Code and data states for **checkout**

Fig. 2. By adjusting two knobs (i.e., orange boxes for code and data), we can achieve (a) execution rollback and (b) code+data checkout. **Execution rollback** is equivalent to altering only the data while keeping the code the same. **Checkout** is equivalent to altering both code/data to exactly a point that existed in the past. In addition to proposing this two-dimensional versioning, we formalize *consistent* code/data states (§5), develop a working prototype (§4 and Appendix A), and evaluate its usefulness in data science tasks (§6 and §7).

on top) to the first commit (i.e., `data=read(data.csv)`) at bottom). Since only the data (i.e., variable) state has changed, users can still see all the code blocks they wrote before. Alternatively, they can check out both code and data, moving both "Head: Variable" and "Head: Code", thereby effectively time-traveling back to the target commit. These two primitives, rollback and checkout, ensure *consistency*; that is, the retained/restored data is equivalent to what would have been produced if (some of) the code had been executed from scratch. We formalize this notion in §5. Our approach is unique compared to the one-dimensional versioning employed by code version control (e.g., git), database checkpointing, etc. While this paper focuses on computational notebooks for clearer presentation, the same design principle applies to a broader range of interactive data science, including Matlab, Stata, SAS, and so on.

To verify the effectiveness of our code+data versioning, we have built an end-to-end prototype, called Kishuboard. Any code+data state manipulation on Kishuboard alters in real-time the actual objects and code served on Jupyter. Likewise, code executions in Jupyter appear immediately on Kishuboard as additional commits, advancing "Head: Code" and "Head: Variable" appropriately with optional branching if necessary. This seamless integration is enabled by our novel component that monitors Jupyter and synchronizes its state according to user actions in real-time (§4).

We evaluate the usefulness of nonlinear data science using Kishuboard through human subject studies (§6). Our quantitative study shows that Kishuboard helps users improve their productivity, especially in intensive workloads when re-execution takes a long time. Our qualitative data shows that users think Kishuboard's features are useful, the UI designs are mostly user-friendly, and they're willing to use or even pay for Kishuboard in the future (§7).

This work makes the following contributions:

Conceptualizing Two-Dimensional Code+Data Exploration To the best of our knowledge, we are the first to propose and implement a new versioning system for interactive data science (Kishuboard) that supports non-linear exploration of both code and data. In contrast, prior work only supports the versioning for code [15, 21, 22, 31]. We show through formative interviews with data scientists the critical need for versioning both code and data states. Using those insights, we introduce the two new primitives: code+data checkouts, and code execution rollbacks. We build a real-time versioning system that supports these two primitives, and integrates with a computational notebook, and which ensures the code and data state consistency and supports the temporal exploration of a two-dimensional code+data evolution history(§4, §5, and Appendix A). Supporting non-linear exploration can help significantly reduce the time for the development of new data science models.

An interface for Two-Dimensional Code+Data Exploration To the best of our knowledge, we are the first to design an interface that supports exploring two-dimensional code+data states. In contrast, prior interfaces have focused on navigating changes to code [15, 21, 22, 31], pain point of notebooks [20, 40, 47, 48, 53], visualizing the change of data frames [16, 49], or the user's behavior when exploring data [13, 16, 23, 30, 39]. The key design elements (informed by formative interviews (§3) and refined using iterative pilot studies) of the interface are mapping the 2-D code+data evolution history to a 1-D graph, from which user can select a previous code/data version, inspect its state and checkout to it, and using methods like execution counter, tagging and search to help users locate the target commit. Extensive human subject experiments demonstrate the utility (i.e., time savings) of the interface, which is more pronounced in the exploration of large models and datasets.

2 BACKGROUND AND WHY NOW

This section briefly describes the advances in data science checkpointing. We use Jupyter [19] as an example considering its impact [3, 12, 36]. This will help readers understand existing pain points (§3) and our system design (§4 and §5).

2.1 Existing System for Interactive Data Science

Jupyter is a web interface running on top of a Python interpreter. Jupyter lets users submit a code block one at a time. The result can be presented in diverse formats, ranging from text to JavaScript-based interactive plots. Leveraging the Python ecosystem, users can import various libraries for data analytics, machine learning, visualization, and so on. In this work, we focus on providing nonlinear data exploration capabilities for this general framework without requiring any changes to Jupyter or any other libraries.

2.2 No Checkpointing in Existing Data Science

Database systems such as PostgreSQL and Microsoft SQL Server create *checkpoints*, periodically saving changes [34, 43]. In contrast, data science systems lack mechanisms for identifying changes, an important premise of checkpointing. Existing databases achieve this through centralized buffer pages. If data is updated, the corresponding pages are marked *dirty* and saved later. Data science systems intentionally omit central data spaces. Instead, they allow individual libraries to manage data using shared memory [4], GPUs [2, 37], and remote machines [35, 55] for high performance. These libraries rarely trace changes in data. As a result, nearly all data science systems, including Jupyter, lacks checkpointing.

2.3 Advances in Data Science Checkpointing

There are recent advances toward checkpointing data science systems (i.e., checkpointing kernel data) [26–28, 32]. The observation is that the standard serialization protocol is insufficient for capturing the state. Instead, object dependencies must also be considered to capture the state correctly [28]. The idea has been validated for a large number (146) of data science classes [27] including in-memory analytics [14, 33], GPU training [2, 17, 37], and distributed computing [35, 55]. The method has been integrated with batch jobs [7] and Jupyter [26].

However, the ability to checkpoint data is insufficient for nonlinear data science. First, we need a principled design for managing code and data states consistently and presenting them clearly. Second, we should validate the usefulness of nonlinear data exploration for actual data science tasks. This paper aims to close this important gap.

3 FORMATIVE INTERVIEW

We met with six data scientists to hear their thoughts on nonlinear data science and how checkpoints for both code and data can be potentially used to support it. These conversations suggest a new versioning mechanism in the two-dimensional code+data space. Compared to existing works that focus only on the pain point of notebooks [20, 40, 47, 48, 53] or the user's behavior when exploring data [13, 16, 23, 30, 39], we also learned lessons about how versioning for both code and data can potentially help them and what's the concerns when it comes to developing a complete version management system for nonlinear interactive data science based on those code and data checkpoints (§3.4).

3.1 Interview Methodology

Group. Five data scientists (F1–F5) were from Meta, Microsoft, and start-ups. They all had PhD in computer science and more than two years of industry experience. We also interviewed with a fourth-year PhD student (F6) studying machine learning.

Procedure. We had a 60-min semi-structured interview with every data scientist. Each interview had three parts. First, we discussed how they used Jupyter. Second, we asked about current pain points. Finally, we shared the idea of using code+data checkpoints to support nonlinear exploration and asked how it could be enhanced.

3.2 How They Use Jupyter

They used Jupyter for three main purposes. The first was exploratory modeling. Using a sample, they would experiment with different parameters (F6), datasets (F4, F6), or models like gradient boosting (F2, F5). Due to resource constraints, final models were usually trained outside Jupyter (F1-F6). The second use case was data loading and transformation. F4 would start a new session, load a dataset, experiment with various data transformation methods, and then save the transformed results to CSV for future use. Finally, Jupyter was used for quick debugging and sanity checks. F3 would copy suspicious code into Jupyter to verify data formats. In all these, Jupyter was a convenient tool for examining multiple alternatives of models, hyperparameters, data transformation, and so on.

3.3 Current Pain Points

The interviewees (F1–F6) shared various inconveniences they had been experiencing. See Table 1 for a summary.

(P1) Jupyter's web interface may lose its connection with the underlying kernel (i.e., Python interpreter) due to network issues or system suspensions. Then, all the (intermediate) data that have been generated are erased and cannot be restored. F5 and F6 typically re-executed cells to recover their work, which was tedious.

(P2) Using Jupyter to compare different alternatives often overwrites data that cannot be restored easily. For example, F4 noted "If I copy a script from somewhere else into Jupyter and execute it, it might override some variables. It's hard to track. You also need to rerun some cells to refresh the variables before proceeding."

(P3, P4. P5) In Jupyter, it is often hard to understand which of the past executions caused the current (abnormal) result. One mitigation mechanism is to print as many variables as possible every time a code block is executed, which is cumbersome. Of course, we cannot print variables for past states. As F1 noted "Many times the code only works for the current moment because notebook is messy and you don't know how to execute the cells in the right order to reproduce the current variables. You have to mentally maintain the order when developing in notebook."

(P6) A session cannot be moved between machines, making dynamic scaling challenging. Running Jupyter for large-scale distributed machine learning is uncommon in the industry.

Table 1. Pain points in Jupyter

Pain Point	Interviewee	Do We Solve
P1: Session suspension deletes data	F5, F6	Yes
P2: Cannot restore overwritten data	F2, F4	Yes
P3: Hard to debug	F3	Yes
P4: No data versioning	F3	Yes
P5: History becomes messy quickly	F1	Partly
P6: Hard to scale resources	F4	Partly

3.4 How Versioning for Both Code and Data Can Help Them

Benefits of Versioning. When we shared the basic idea of supporting nonlinear data science through versioning both code and data, all interviewees agreed on its potential usefulness. Specifically, many of the current pain points (P1–P4) can almost be directly solved by the new capability. P5 can be partly addressed as users can browse their exploration history and the variables/data from the past states. P6 can be mitigated as checkpointing enables moving Jupyter sessions between machines, allowing dynamic resource scaling.

Further Suggestions. In addition to excitement, the interviewees shared suggestions for further improvements. We summarize this feedback in Table 2, and highlight a few novel ones as follows. S1–S3 suggested independent versioning of code and data. F1 mentioned an interesting use case in which one needs to keep the code but revert variables: "There's a mistake in my code, and I need to revert my variable state from the mistake, and then correct and rerun the code after it." F2 brought up another case: only the input data changes while keeping the code the same.

Some expressed concerns that a commit history may become too complex as exploration continues. This may require additional tools for easier exploration of histories. Also, they mentioned a need for tracking changes in a specific variable. Moreover, comparing (i.e., diff-ing) two commits can be useful for easier understanding.

Table 2. Suggestions for nonlinear data science

Suggestion	Interviewees	Our Approach
S1: Keep code and revert only variable state	F1, F4	} Checkout code and kernel states separately
S2: Distinguish the changes in data sources and code	F2	
S3: Show code/variable change history separately	F2, F4, F5	1-D timeline for 2-D state transition
S4: Concern: Too many commits for one notebook	F1, F4	Context-aware auto-folding (of commit history)
S5: Show the lineage of variables	F1, F2, F3	Search: highlight commits that modified the variable
S6: Compare two commits	F2	Independent `diff` of code and variable
S7: Different commit granularity for different use cases	F2	
S8: Cherry picking variable from a different branch	F1, F6	Can be supported, but not implemented in this work
S9: Sharing, merging, collaboration	F1, F2, F4, F5	

4 SYSTEM OVERVIEW: USE CASES, USER INTERFACE, AND ARCHITECTURE

In this section, we describe how we design a new user interface (UI), called Kishuboard, to address the suggestions from Table 2 (i.e., S1-S6). Specifically, we explore specific use cases (§4.1), and derive a UI for supporting those use cases (§4.2). Finally, we describe how Kishuboard interacts with the underlying Jupyter system (§4.3).

4.1 Potential Use Cases

Manage Alternatives. Alice, a data scientist, completed data transformation, feature engineering, and model selection. During later conversations with a colleague, a new feature engineering method occurred to her, and she wants to try it. Normally, this would require overwriting her data, forcing her to reload the entire dataset and re-execute cells if she needs to revert to her original approach. With Kishuboard, Alice can create a new branch to test the alternative method without losing her current progress. If she decides to return to her original method, she can easily switch back to the previous branch, allowing her to explore multiple approaches simultaneously and efficiently without risking errors.

Understand What Happens. Bob needs to share his work with a colleague or revisit his notebook after some time. Without a tool like Kishuboard, understanding the order in which cells were executed can be challenging, especially since cell execution can happen out of order [15, 21, 22, 31, 52]. This makes it difficult to reproduce results or trace the logic of the notebook. With Kishuboard, Bob and his colleague can view a detailed execution log that shows the order and timestamps of every cell that was run to understand the notebook's history. Although the above feature is also supported by other notebook extensions [15, 21], what's unique to Kishuboard is that additionally it also allows Bob to perform reverse debugging by tracking how variables change across different cell executions, avoiding the need to repeatedly re-execute cells and print outputs to figure out where things went wrong [6].

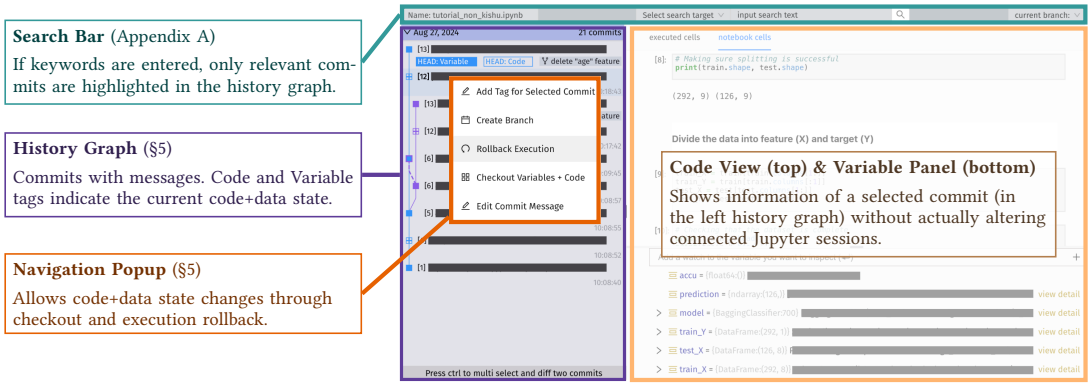


Fig. 3. Kishuboard user interface. The history graph (purple box) shows past commits with code+data tags, allowing users to quickly grasp the current state of computational notebooks (e.g., Jupyter). The code and variable panes (yellow box) display the information of a selected commit (in the history graph). From any past commit, users can load data only (i.e., execution rollback) or load both code and data (i.e., checkout) using the navigation popup (red box). Then, the Code and Variable tags move appropriately.

Quick Work Recovery When Restart. Computational notebooks run on top of the underlying kernels (e.g., Python, R). Kernel crashes, restarts, and disconnections are common issues that data scientists face during their work [6, 9, 29]. Imagine Carol is working on a complex notebook, and suddenly, her kernel crashes. Without Kishuboard, she would have to re-execute all cells from the beginning, which is not only time-consuming but also prone to errors, especially if cells were executed out of order [41, 42, 44, 52]. With Kishuboard, when the kernel restarts, Carol can quickly recover her work by checking out the latest commit. This feature ensures that all her progress is preserved, and she does not have to worry about manually re-executing cells in the correct order. Kishuboard 's ability to handle these interruptions smoothly allows Carol to maintain her workflow with minimal disruption.

4.2 User Interface

The design goal of Kishuboard is to let users easily manipulate the current state of computational notebooks through two independent knobs (i.e., code and data) while allowing them to easily understand how the notebook states have been changed over time. We aim to achieve this high-level goal by introducing minimal changes to commonly used UIs for intuitive understanding and easy adoption.

Components. Kishuboard UI comprises four main components: the code view (right), the history graph (left), the search bar (top), and the variable panel (bottom). The **code view** presents the notebook's state at a specific point in time, displaying the content of various cell types (e.g., code cells, markdown cells) along with their outputs. Users can toggle the code view to an alternative mode that shows the sequence of executed cells.

The information in the code view corresponds to a commit chosen by the user, while the **history graph** shows the relationships among commits. Users can navigate through the history graph to browse commits and select an individual commit for details. The design choices for the history graph will be elaborated in §5.

Finally, the **search bar** offers searches based on commit content, commit message, and names of the variables modified by the commit. Commits that match with the search are highlighted in the history graph. One specific use case for search is to find all commits that changed a specific variable: Users can track how the variable changes by inspecting those commits highlighted as search results. The **variable panel** shows the state of variables at a selected

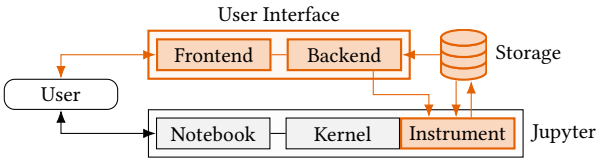


Fig. 4. Kishuboard's system architecture consisting of user interface, storage, and instrument highlighted.

commit. This panel provides details such as the variable name, type, and a string representation of its value, along with extended representations for special types, such as table formatting for data frames. Users can also expand each variable to inspect its nested attributes.

Two Knobs: Code and Data. Unlike other systems (e.g., git), Kishuboard lets users navigate in the two-dimensional code+data space. Specifically, the users can load data (and variables) from a past commit while retaining the current code, effectively achieving execution rollbacks. Using Kishuboard's UI (Fig 3), users can perform this execution rollback by right-clicking a target commit and selecting "Rollback execution". The actual Jupyter state changes immediately as Kishuboard lets it load variables from the target commit. Jupyter execution counters—displayed by each cell to indicate orders—update appropriately. Likewise, the users can load both code and data states from a past commit by selecting "Checkout Variables + Code", which amounts to time-traveling the entire system state. This is analogous to checkout in traditional versioning systems (e.g., git).

4.3 System Architecture

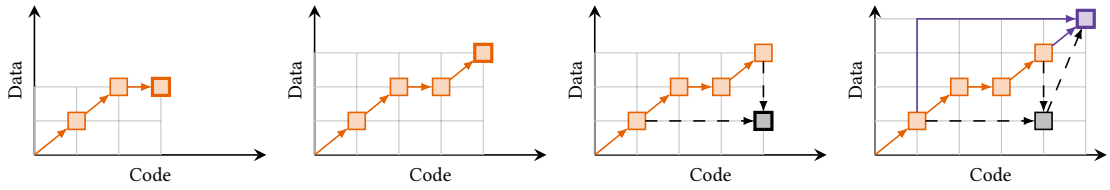
To support the UI above, our system is designed to have three main functions: (1) **Collecting Information**: When Kishuboard is enabled on a notebook, it attaches an *instrument* to the Jupyter kernel. This instrument detects cell executions and captures all relevant notebook state information, including cell content, cell outputs, and variable values. It then persists this data in Kishuboard's *storage*.[1] This storage also maintains additional metadata, such as commit histories and branch positions. (2) **Browsing Code and Data**: The *UI* of Kishuboard reads from the storage and preprocesses this information in its *backend* server before displaying it on the *frontend* web interface. When users interact with various UI components—such as selecting a commit or conducting a search—the UI requests the additional data and updates the displayed state accordingly. (3) **Altering Notebook State**: Certain interactions, such as checking out a commit or rolling back executions, require changes to the Jupyter kernel and notebook. In these cases, Kishuboard's UI sends commands to its instrument, which reads the necessary information from storage and modifies the kernel and notebook states as needed.

Implementation. Kishuboard's UI is a web application consisting of a frontend written in TypeScript with React and a backend written in Python with Flask. Kishuboard's storage is an embedded database library written in Python. We integrate Kishuboard's instrument onto Jupyter based on its extension framework [18].

5 FORMALIZING CODE+DATA VERSIONING: MODEL, CONSISTENCY, AND VISUALIZATION

Users need the ability to recover code and data separately (Table 2), but certain recovery actions can confuse the users and unintentionally lower productivity. To address this, we first formally define the data model (§5.1) and its state transitions (§5.2) to identify cases where checkouts can be problematic (§5.3). To overcome these issues, we introduce a

[1]In our implementation, only the change of variable values are stored to ensure efficiency, the mechanism is similar to the one used in [27]



(a) Initial commits V_1, V_2, V_3 (b) Executing a cell V_4 (c) Rolling back (C_4, D_1) (d) Executing a cell V_5

Fig. 5. An example of navigation in 2D versions, creating a history of versions. Thick rectangles denote head versions.

consistency property as a safeguard against such problematic actions (§5.4). Finally, we present a dimension reduction strategy for visualizing 2D versions in a 1D space on the user interface (§5.2).

5.1 Data Model

Kishuboard helps users manage "code+data" versions, which broadly encompass notebook and variable states. This section formalizes what Kishuboard regards as code, data, versions, and commits.

Code and Data. On the high level, the code state consists of notebook cells (e.g., code and Markdown cells) and outputs, while the data state consists of a variable map (i.e., a mapping from variable names to their values) as well as execution log and data history. This execution history is a list of executed code cells (i.e., a list of strings) that produces the corresponding variable map. Note that, in contrast to the `ipynb` file extension, which stores the execution counters displayed on the left of code cells in the file, Kishuboard considers these execution counters as a part of the data state (precisely execution log), not the code state, which helps identify inconsistent states (§5.3).

Let C_i be the i-th code state and $D_j = (X_j, H_j)$ be the j-th data state consisting of variable map X_j and execution history H_j. An execution history is a list of strings $H_j = [h_j[1], h_j[2], \dots]$ where $h_j[n]$ denotes the n-th executed cell.

Versions. A version is a pair of code and data state; that is, $V_k = (C_i, D_j)$. While later sections will discuss the recovery of code and data separately within Kishuboard, code, and data states naturally progress independently in existing computational notebook settings without Kishuboard. For one, when users *edit* any of the notebook code or Markdown cell, the code state progresses while the data state remains the same, from $V_k = (C_i, D_j)$ to $V_{k'} = (C_{i'}, D_j)$. On the other hand, when users *execute* a code cell c, they may update both code and data states by generating a new cell output and mutating the variable map from $V_k = (C_i, D_j)$ to $V_{k'} = (C_{i'}, D_{j'})$. In this latter case, the execution history will include the newly executed code cells, $H_{j'} = H_j \oplus [c]$, where $\oplus$ is the concatenation between two lists.

Commits. A commit is a version that Kishuboard persists. By default, Kishuboard creates and persists a version into a commit after every cell execution. In addition, Kishuboard also allows users to create a commit manually. Because Kishuboard does not persist all versions, commits are a subset of all existing versions. We denote $\mathcal{V}$ as the set of all versions and $\mathcal{U}$ as the set of all commits (i.e., $V_i \in \mathcal{U}$ implies that Kishuboard has persisted the version V_i).

5.2 Navigating in 2D Versions

Apart from editing the notebook and executing a code cell, notebook users may navigate to different versions through Kishuboard by checking out a commit and rolling back execution that replaces the source's code and/or data state with the destination's. Although a sequence of navigation forms a history of notebook versions residing in a 2D space, Kishuboard visualizes the committed versions in a 1D commit history graph.

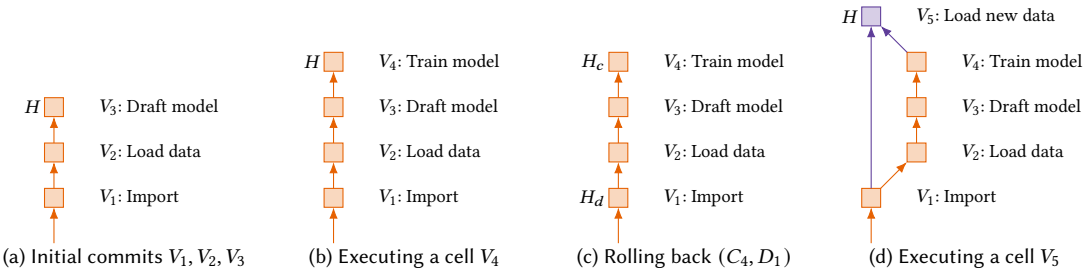


Fig. 6. An example of navigation in 1D commit history graph, creating a history of versions. H_c and H_d denote the current head's code and data version, respectively; when they are on the same version, we use H to denote them together.

2D Versions and History. Versions are 2-dimensional by definition; a version V_k is described by a coordinate of C_i and D_j. For example, in Fig 5a, Bob, a data scientist, has created three committed versions $V_1 = (C_1, D_1)$, $V_2 = (C_2, D_2)$, and $V_3 = (C_3, D_2)$ from his notebook. When he executes a cell to train his model, he will create another committed version $V_4 = (C_4, D_3)$. Apart from these four versions, the notebook could be in any combination of code and data states such as (C_1, D_2) or (C_3, D_3), denoted as a grid in Fig 5b. Theoretically, it is possible that Bob could roll back execution from $V_4 = (C_4, D_3)$ to (C_4, D_1) (Fig 5c).

Each committed version V_k has a code parent $P_c(V_k)$ and a data parent $P_d(V_k)$ which indicates the previous respective state. Both parents may belong to the same or different committed versions. For example, V_3 is both code and data parent of V_4 because $V_3 = (P_c(V_4), P_d(V_4))$. On the other hand, when Bob reloads his data after rolling back execution, he would create a commit V_5 with a code parent $P_c(V_5) = C_4$ and a data parent $P_d(V_5) = D_1$. The collection of versions and their parents form a history $(\mathcal{U}, P_c, P_d)$, embedded in the 2D code+data space.

Visualization in 1D Commit History Graph. Instead of displaying commits using the 2D code+data space, Kishuboard opts to visualize the history using a 1D **commit history graph** that represents commits using nodes and their parent relationship using edges. Fig 6 shows the commit history graph equivalent to the 2D history in Fig 5. By doing so, we simplify the user's learning curve due to its resemblance with familiar Git interfaces [10, 11, 46] that allows scrolling and presenting other commit information like commit messages and timestamps.

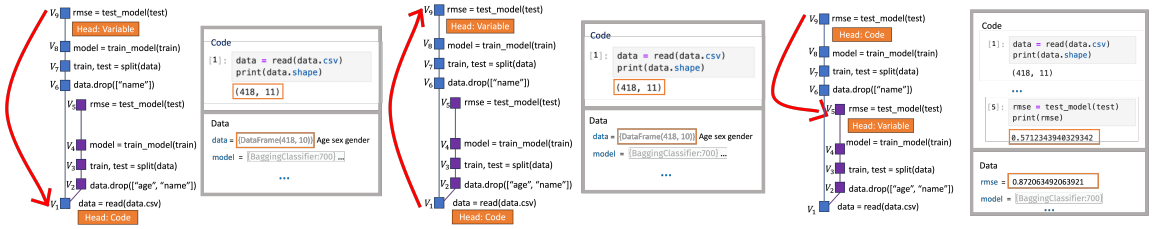
In addition, this commit history graph denotes the head commit $H = (H_c, H_d)$ (i.e., the current version) by labeling the corresponding commit(s). If the current head's code and data states belong to different commits, the commit history graph labels the two commits accordingly (e.g., Fig 6c). As a result, the commit history graph displays a commit with two different parents (e.g., V_5 in Fig 6d) by connecting it directly to its parents.

5.3 Problematic Checkouts

Theoretically, Kishuboard could allow checking out any pair of code state C_i and data state D_i independently; however, we observe three types of problematic checkouts that create inconsistent states, as illustrated in Fig 7.

Problems When Checking Out Only Code. In Fig 7a, consider a scenario where only the code is checked out to V_1 while the data is still in V_9. Based on the code, users misunderstand that `df` holds the data from the original `data.csv`. However, since the kernel data is still V_9, the `name` feature is dropped from the original `df`.

Problems When Checking Out Future Data. Similarly, checking out data from V_1 to V_9 but keeping the code in V_1 will lead to the same inconsistency as checking out only code.



(a) Checkout Only Code. (b) Checkout Future Data. (c) Checkout Unrelated Data.

Fig. 7. Three problematic checkouts that break consistency. The inconsistency of code and data is highlighted in red rectangles. **(a) Checkout Only Code**: the code(including cell output) is checked back to reflect data in V_1, but the data is still in V_9. **(b) Checkout future data**: the data is checked out to V_9, but the code still reflects data in V_1. **(c) Checkout Unrelated Data**: the data is checked out to V_5 on another branch, but the code still reflects the current branch's data.

Problems When Checking Out Unrelated Data. Alice is at V_9, and she checks out only the data to V_5 from a different branch in history. This is also confusing because the code is still at V_9, and from the code, users assume the `age` feature is never dropped, and the model is trained with the `age` feature, with the `rmse` of `0.5123`; however, this is not the case for our variables in kernel, which is already in V_5. The nature of the problem is that V_5 and V_9 are on different branches, so their histories are unrelated.

5.4 Consistency and Safe Checkouts

A version of code and data states $V_k = (C_i, D_j)$ is **consistent** if and only if the cell outputs in C_i agree with the results of execution according to the execution history H_j. More formally, for all executed cell $c \in C_i \cap H_j$, let n be the time the cell c is executed (i.e., $h_j[n] = c$), $V_k = (C_i, D_j)$ is consistent if and only if $\text{Output}(c) = \text{Exec}([h_j[1], h_j[2], \ldots, h_j[n]])$. Here $\text{Exec}(H)$ is the result of execution over a list of code cells H and $\text{Output}(c)$ is the output of the cell c.

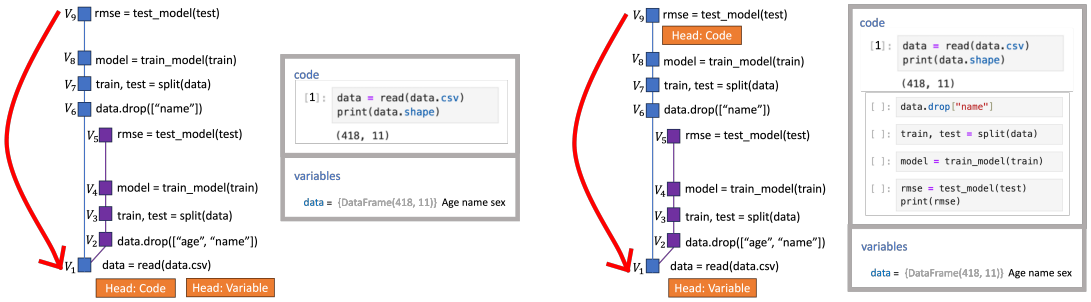
Theorem 5.1. *All commits $V_k \in \mathcal{U}$ are consistent.*

Proof. We prove the statement by induction. Recall that the empty notebook version $V_0 = (C_0, D_0)$ where C_0 contains no cells and D_0 contains no variables is consistent because the set of executed cells is empty $C_0 \cup H_0 = \emptyset$.

All commits are subsequent versions of the empty notebook version by editing or executing notebook cells; therefore, it suffices to prove that editing or executing notebook cells from a consistent version $V_k = (C_i, D_j)$ makes a consistent version $V_{k'} = (C_{i'}, D_{j'})$. If a cell $c \in C_i$ is edited, c becomes a non-executed cell in $V_{k'}$; $V_{k'}$ remains consistent because of V_k consistency on other executed cells. If a cell $c \in C_i$ is executed, it becomes an executed cell $c = h_{j'}[n] \in H_{j'}$ and its output is updated with the execution result $\text{Output}(c) = \text{Exec}([h_j[1], h_j[2], \ldots, h_j[n]])$, which satisfies the consistency condition. Hence, editing or executing notebook cells preserves consistency. □

Problematic checkouts (§5.3) are confusing because they can induce inconsistent code+data states. Checking out only code may introduce inconsistent cell outputs from a different sequence of cell executions. Checking out future data may contain missing cell outputs. Lastly, checking out unrelated data on a different branch may admit execution results that conflict with cell outputs. To prevent potential confusion, Kishuboard implements a safeguard that disallows these unsafe checkouts. As a result, Kishuboard allows two types of safe checkouts that guarantee the consistency:

Checking Out Both Code and Data. Users can safely check out both code and data states from any commit. Thanks to Theorem 5.1, checking out both code and data from a commit always produces a consistent version. For example, in



(a) Checkout both code and data.

(b) Checkout past data.

Fig. 8. Consistent checkouts

Fig 8a, when we are on V_9 and want to continue exploring the case where age feature is dropped, we can check out both code and data to V_5 and continue from there.

Checking Out Past Data. Users can safely check out a data state from a past commit (i.e., rollback execution). For example, in Fig 8b, at $V_9 = (C_9, D_9)$, the data scientist checks out the data state to D_1 and keeps the code state. Checking out only data generates a new version (C_9, D_1). With C_9 kept there, the user does not need to rewrite cells that are shared between different explorations.

Theorem 5.2. *If $V_k = (C_i, D_j)$ is consistent, the version after checking out a past data $V_{k'} = (C_i, D_{j'})$ is consistent.*

Proof. A past data state $D_{j'}$ contains an execution history that is a prefix of the current one $H_{j'} \subseteq H_j$. For all executed cell in the checked out version $c \in C_i \cap H_{j'} \subseteq C_i \cap H_j$, its output matches with the execution result $\text{Exec}([h_{j'}[1], h_{j'}[2], \ldots, h_{j'}[n]]) = \text{Exec}([h_j[1], h_j[2], \ldots, h_j[n]]) = \text{Output}(c)$. □

6 USER STUDY DESIGN

We aim to answer two core questions: (1) Does Kishuboard improve productivity in data science tasks? (2) Are the Kishuboard's UI designs intuitive and easy to interact with? We study these with a user study with 20 students. These students are randomly assigned to the experimental group with Kishuboard and the control group without Kishuboard (§6.1). For both groups, we provide an identical testing environment with the same tutorials, data science tasks, reference code, and instructions (§6.2). We observe how the users perform the task, measure the time they spend on each part, and collect feedback through exit surveys (§6.3). Lastly, we discuss our efforts to minimize the gap between this lab experiment and real-world data science tasks (§6.4). The University Ethics Review Board approved human subject experiments and the formative study.

6.1 Participants: 20 Students with Backgrounds in Data Science and Jupyter/Python

We recruited 20 students from a course covering the intersection of machine learning and data management: 1 undergraduate student and 19 graduate students. The same amount of extra credits was given as an incentive for participating in this study, regardless of their performance. With "Artificial Intelligence" as a course prerequisite, all participants had backgrounds in data science workflows such as extract-transform-load, feature engineering, model training, etc. Also, all the participants already had completed course assignments related to Jupyter and Python.

Table 3. Participant group assignments. 20 students with backgrounds in Python/Jupyter were randomly assigned to the experiment group (w/ Kishuboard) and the control group (w/o Kishuboard). The experimental group was divided into two subgroups (KL and KS). Each participant in KL was assigned compute-intensive tasks, and each in KS was assigned lightweight tasks. Likewise for the control group: CL and CS for compute-intensive and lightweight tasks, respectively.

Group Name	Kishuboard Used?	Workload Scale (Running the entire notebook end-to-end takes)
KL	Yes	Compute-intensive (332 secs)
CL	No (control)	Compute-intensive (332 secs)
KS	Yes	Lightweight (80 secs)
CS	No (control)	Lightweight (80 secs)

The participants were randomly assigned into 4 (sub-)groups as in Table 3. All four groups performed the same tasks, but differed only in the use of Kishuboard and workload scales. The only difference between lightweight and compute-intensive tasks was artificially injected delays for certain cell executions (e.g., data loading, training models); however, the contents of the datasets and trained models were exactly the same.

6.2 Preparation: Environments, Training, and Task Design

Testing Environment. We pre-installed both Jupyter and Kishuboard on the testing laptop. The laptop was a MacBook Pro with 16 GB memory and a 2.3 GHz Intel Core I7 processor. We extended the laptop with an LG 34-inch 21:9 UltraWide 1080p monitor for the user study. We 1:1 split the screen into two parts, one for Jupyter and the other for Kishuboard; participants could see both simultaneously.

Kishuboard Introduction Video. We prepared a 4-minute video to introduce Kishuboard and demonstrate its features. The video gave users a basic idea of Kishuboard and how it worked. Users in KS and KL watched this video at the beginning; those in CS and CL watched it after the formal tasks.

Tutorial. Tutorial notebooks were prepared to help participants recall Jupyter (for all groups) and try Kishuboard before starting actual (i.e., formal) tasks (only for KL and KS groups). The tutorial notebooks had a structure similar to the formal tasks for an easier transition, but their semantics and difficulty were different. For Kishuboard groups (KL and KS), tutorial notebooks had extra modules for trying Kishuboard beforehand. This trial session was less than five minutes; thus, their exposure to Kishuboard was relatively limited (compared to Jupyter and Python).

Formal Tasks. We prepared two sets of tasks: Workflows I and II. Each workflow contained multiple (smaller) tasks.

Workflow I: Build data science models This mimicked how data scientists would build models to predict Titanic fatalities. Users were instructed to follow a typical data science workflow and choose the methods from a provided list of codes for data loading, visualization, cleaning, feature engineering, model training, and evaluation.

Task 1: Try alternative They were asked to change the feature engineering method and retrain the model. This task requires nonlinear exploration because some features were dropped previously.

Task 2: Report old & new values They were asked to report the old and new model accuracies (RMSE).

Task 3: Retrieve old variable Finally, they are asked to export both old and new models for future inspection.

Workflow II: Recover work from bugs and system crashes While sequentially executing the code blocks in a notebook, participants would find that a bug has corrupted `data` variable.

Task 4: Identify bug They were asked to find the root cause of the bug.

Task 5: Recover from restart After fixing the bug, they were asked to restart the kernel. This resembles a system crash. Participants were then asked to restore their work.

To make the notebooks more realistic, all code snippets were taken from Kaggle prize-awarded solution notebooks, with minor modifications of variable names to ensure compatibility and simplify the context. Users were asked to open and read instructions for one task at a time. This was to simulate the situation where they progressively develop their notebook. Users were free to modify all notebook cells; they were only asked to complete each task correctly.

6.3 Procedure

Experiment Groups w/ Kishuboard (KS, KL). We first presented a Kishuboard introduction video to participants. participants were then asked to study the tutorial notebook. Later, they were instructed to work on the two formal task notebooks. Lastly, they filled out an anonymous survey.

Control Groups w/o Kishuboard (CS, CL). Instead, participants were first asked to study the tutorial notebook. Then, they were instructed to work on the two formal task notebooks. Afterward, we presented Kishuboard introduction video to participants. Lastly, they filled out an anonymous survey.

While the users were working on the formal tasks, we measured the time they spent on each task. The user study lasts for about an hour across all participants.

6.4 Our strategies toward realistic data science environments

While our experiment setup may not exactly replicate real-world data science workflows, we employ various mitigation strategies to close the gap. Table 4 describes potential differences between real-world data science and our lab environment (the first and the second columns), and present our strategies (in the third column). For example, real-world data scientists would convert their high-level goals to code by themselves. In contrast, our user study provided participants with various code snippets to help them complete tasks in a limited amount of time (i.e., one hour). To narrow this gap, we provided multiple alternative code snippets. Participants could mimic the process of "writing code" with "their own choice." Another limitation would be users' lack of experience with Kishuboard. To mitigate this, we provided training with a tutorial notebook.

Table 4. Our strategies to close the gap between real-world data science and our user studies.

Real-world data scientists	Discrepancy from Real-world	Our user study design to close the gap
Write notebook code by themselves	Code is provided by us	Multiple code snippets to choose from (§6.2)
May have backgrounds with their tasks	Limited time to understand our tasks	Tutorial notebooks in the same structure with easier tasks (§6.2)
More familiar with Jupyter and other tools	No experience with Kishuboard	Tutorials have step-by-step instructions for Kishuboard (§6.2)
Test various models and datasets	Only a few scenarios can be tested	"compute-intensive" and "lightweight" tasks for diversity (§6.3)

7 USER STUDY RESULTS

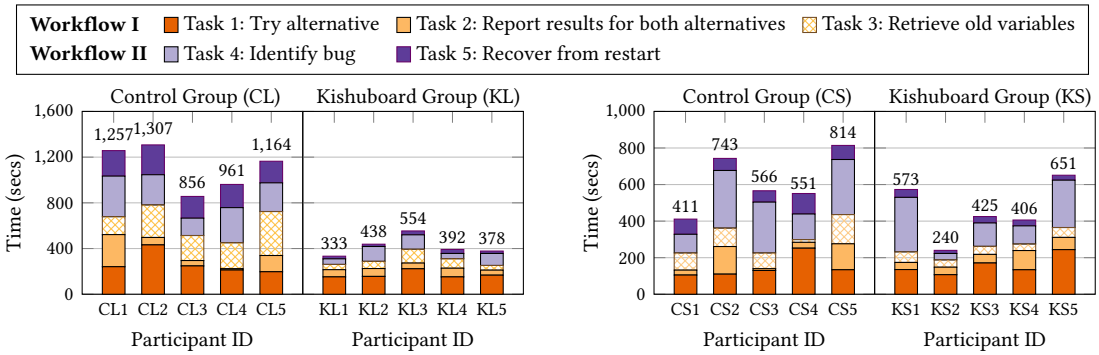
We conduct a user study to evaluate Kishuboard as a nonlinear interactive data science tool. The study shows that:

(1) The proposed code+data space versioning boosts productivity. Kishuboard helped users complete tasks more quickly. The improvements were more significant (62% faster on average) for compute-intensive workloads. For lightweight workloads, the improvement is 25.61% on average. (§7.1)

(2) Kishuboard's features are useful for data science workflow. All the features (i.e., checkout code+data, checkout past data, locate commit by exec counter, locate commit by searching, browse exec history and variable info) were considered "very useful" or "useful" and were used by most participants. (§7.2)
(3) Our proposed UI design is easy to use but has room for improvement. Most participants considered it easy to browse commits and locate the ones they needed. They also considered our UI easy for tracing the variable changes and understanding execution history. (§7.3)
(4) Participants were willing to continuously use or even pay for Kishuboard. 19/20 participants were "willing" or "very willing" to use. 13/20 participants were willing to pay at least $5 a month. (§7.4)

7.1 Code+data space versioning boosts productivity

We quantify Kishuboard's productivity benefits by comparing the performance of Kishuboard groups (i.e., KL and KS) and that of the control groups (i.e., CL and CS) on five different tasks (i.e., Tasks 1–5). The results are shown in Fig 9. We observe that Kishuboard offered significant productivity benefits, especially when the workload was more compute-intensive (Fig 9a). This was because Kishuboard allowed users to explore alternative paths without re-executing some of the code blocks. They could directly check out or roll back executions to retrieve any previous data and resume from that point. This time-saving benefit was lower for lightweight tasks, unexpectedly, because there is less time penalty in manually simulating nonlinear exploration. The time-saving benefit was the biggest for Task 5 (Recover from restart) because re-executing every cell from scratch is time-consuming. Such a manual restoration strategy can only work for small-scale workflows with a relatively short history (like in our user study), which will become extremely challenging in real-world data science.



(a) Performance on compute-intensive workloads

(b) Performance on lightweight workloads

Fig. 9. User performance on Tasks 1–5. Kishuboard helped participants complete tasks more quickly. The improvements were more significant for compute-intensive workloads (left). For lightweight workloads (right), some people could quickly achieve nonlinear exploration by re-running code from scratch, reducing the advantage offered by Kishuboard.

The average treatment effect was statistically significant even considering the variance in individual performance. Specifically, the overall performance (i.e., the sum of the times spent on Tasks 1–5) of the KL group was higher than the CL group with a p-value 4.17×10^{-5}; the performance of KS was higher than CS with a p-value of 7.9×10^{-2}. We also manually confirmed, based on video recordings of user interactions, that those performance variances were caused by natural data exploration, not by any unexpected reasons (e.g., system crashes). For example, in Task1 (Try alternative), KS4 and KS5 lost their code by checking out both code and variable instead of checking out only data. As a result, they

needed to re-choose the same code twice. We also observed that different CS and CL participants selected different strategies to perform their tasks; this had an interesting implication that the additional data operations they performed on an earlier task often aided their later tasks. For example, CS4 was relatively slow in Task 1 but was much faster in Task 2 and Task 3. It was because this user duplicated every cell and renamed every variable to avoid overwriting, which slowed down trying model alternatives but simplified model comparison and retrieving old models. Some other participants, like CL2, CL3, CS1, and CS3, did not overwrite the old RMSE when printing the new one by duplicating just the model evaluation cell, helping them complete Task 2 quickly.

7.2 Kishuboard 's features are useful for data science workflow

7.2.1 Participants' direct feedback was overall positive. We asked participants (in KL and KS) to rate the usefulness of each Kishuboard's feature. The results are reported in Fig 10. Most users found both "checkout only data" and "checkout both code and data" to be "very useful" when building data science models. This is consistent with their actual behaviors, as we describe shortly. Also, most users thought that "searching" and "using execution counter to locate commit" were useful for finding a commit. Participants also mentioned as additional notes that checkout could be useful in their real use cases. One person wrote: when they "want to try out different ways" or "find myself already executed some wrong code messing up the variable states" because checkout helps them to "not losing data and code from the previous model" and "not need to re-execute from scratch."

Search was marked by 7 users as "very useful" for the debugging workflow (second notebook), where they needed to understand how variables evolve. According to our observation, when users knew the execution counter for their target commit (i.e., an increasing sequence number assigned to each executed cell to indicate completed cells), they could locate it more easily (Task 1). Otherwise, they had to search for the commit they wanted using some criteria (e.g., in Task 4, search by variable changes). In addition, a user noted as an additional comment: "in my use of Jupyter

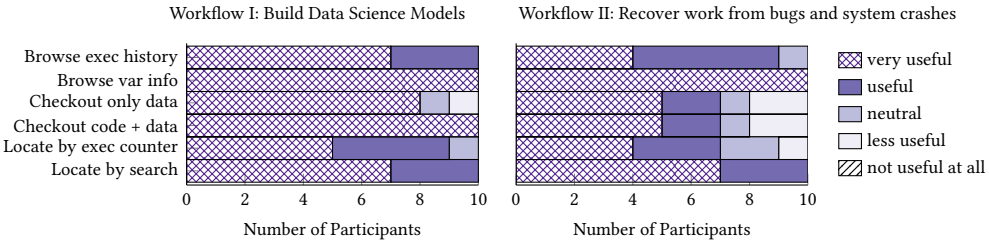


Fig. 10. Participants' perceived usefulness of each feature across different workflows. All the features are considered "very useful" or "useful" by most participants.

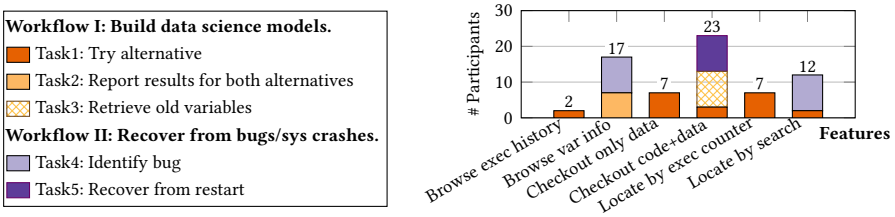


Fig. 11. Number of participants using features for each task. All features are used, with different tasks focusing on different ones.

notebooks it becomes hard to find a particular cell in a large notebook, in particular when I use Jupyter notebooks for running code generation models in Google [Colab]. Searching for a particular cell would be my best real use case."

7.2.2 Kishuboard's features were all frequently used. In addition to directly asking for ratings, we also examined the actual uses of various features offered by Kishuboard. For this test, we manually revisited the screen recordings of participant interactions with Kishuboard. Fig 11 shows how many times each feature was explicitly used by the participants. Seven of the ten participants from the Kishuboard groups used both "checkout past data" and "checkout both code and data" in the "Build data science models" workflow. Specifically, most participants used "checkout only data" for Task 1 to start a new exploratory branch while keeping the code. Everyone used "check out both code+data" for Task 3 to hop between branches. This result confirms that Kishuboard offers the types of navigation that could not be achieved by one-dimensional versioning, and that different kinds of two-dimensional navigation were employed freely by participants to boost their performance.

7.3 Our proposed UI design is easy to use but has room for improvements

We asked the participants about how easy it was to use each UI component (Fig 12). First, most features were rated "very easy" or "easy" to use from at least 80% of the users. Second, 60% of the respondents felt "neutral" or "hard" to understand the difference between dashed lines and solid lines, which represent code and data parents in the history graph. We believe we can largely address this usability issue by better highlighting in our tutorial how notebook states may proceed in the code+data space.

Some participants also commented on potential UI improvements, especially in the search bar, display, and integration. One respondent suggested extending the search query language to support deeper semantics such as attributes/fields of variables. Another participant suggests integrating Kishuboard into Jupyter.

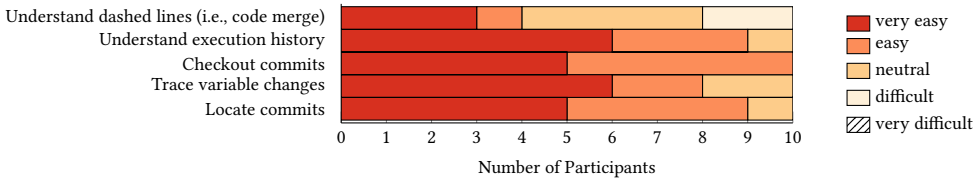
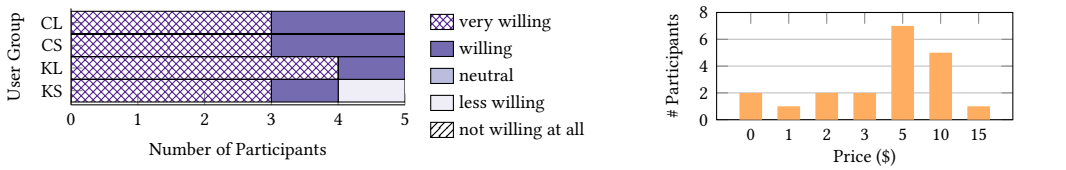


Fig. 12. Participants' perceived easiness to use/understand UI design. Except for the "dashed line," all the other designs are considered "easy" or "very easy" to use/understand by most users.

7.4 People are willing to use or even pay for Kishuboard

We evaluated Kishuboard's overall usefulness by asking two types of questions: are you willing to **use** it in the future, and how much would you **pay** to use it? Fig 13a presents users' willingness to use Kishuboard for their data science tasks. The results indicate that most participants were "very willing" and "willing" to use Kishuboard. One participant from the KS group said, "I took the 'Machine Learning' last semester, in the MPs, I need to restart and re-execute to get over-written variables frequently. I wish I had this tool." Another participant from the CS group said, "I used to be a data scientist in the industry, and I would say this tool definitely helps my work, as we need to bounce back and forth a lot."

We also asked participants how much they would be willing to pay for Kishuboard if, for example, it was a feature of Google Colab+ (Fig 13b). For those who gave a price range as their answer (e.g., $5-$10), we report the smaller amount (e.g., $5). The most common choice was $5/month, with 7 participants indicating the amount. Interesting, there were 6



(a) Participants' willingness to **USE** Kishuboard

(b) Participants' willingness to **PAY** for Kishuboard

Fig. 13. User's willingness to use and pay for Kishuboard. The figure shows most participants are willing to use and pay for Kishuboard as an extension to Google Colab.

participants willing to pay at least $10/month for Kishuboard. Among the two participants who weren't willing to pay for Kishuboard, one explained "I would not be willing to pay. I already subscribe to Google Colab Pro, and if Kishuboard was added as a premium feature it would entice me to continue paying the subscription fee." From this comment, we believe the participant's unwillingness could be partly attributed to the upselling pricing scheme rather than a lack of inherent value in Kishuboard. The particular product name "Google Colab" was mentioned in our user study to aid understanding; however, this project is not affiliated with or sponsored by any for-profit organization.

8 RELATED WORK

This section lists related works in assisting data science workflow on computational notebooks (Table 5).

Notebook Version Control Systems. While Git [31] can store and manage computational notebooks as source files, many version control systems are specialized for notebook workflows. Verdant [21, 22] supports notebook file version control on a finer granularity such as cell-level version control. Code Gathering [15] allows users to automatically extract the smallest necessary code subsets to recreate their chosen outputs. Although these systems help users select the relevant code from a potentially messy notebook execution, they do not support recovering data, forcing users to re-execute the code to reach a desired kernel state.

Branched Presentation of Cells. Several works [40, 47] design a tree-like navigation panel that represents the branched logical structure of a notebook. Other works [20, 48, 53] present cells from different branches in a paralleled view. For example, in Variolite [20], users can craft alternative code snippets and explore different combinations through revision trees. Like version control methods, these methods help users identify codes relevant to their current exploratory branch; however, to resume the work in a different branch, users still need to re-execute the code.

Fork Kernel. To support quick data recovery between different branches, Fork it [54] forks a new notebook for each alternative. However, this method has two problems. First, it is not scalable because forking the kernel per alternative

Table 5. Comparision between Kishuboard and other systems for supporting non-linear exploration.

Approach	Mechanism
Notebook version control system [15, 21, 22, 31]	Record code version or execution order (cannot recover data).
Branched presentation of cells [20, 40, 47, 48, 53]	Present branched structure of code (cannot recover data).
Fork kernel [54]	Fork kernel for alternatives (not scalable, cannot recover past data).
Record variable change history [16, 49]	Visualize variables' evolution history (no checkout).
Checkpoint and restoration [8, 27, 28, 32]	Store data science objects/variables (not considering UI/UX).
Ours (Kishuboard)	**Two-dimensional code+data space versioning**

can be expensive. Second, it does not support forking from a past state; in other words, the user cannot revert the data state to try an alternative, so they must anticipate the forking beforehand. In contrast, Kishuboard implements an incremental checkpointing as discussed in [27] and empowers users to explore both past and alternative states.

Record Variable Change History. [16, 49] record the variable change history and support comparison between different variable versions. However, they offer no solution to recover the variable to resume notebook execution. Beyond supporting variable comparison as well as searching for variable changes, Kishuboard also allows users to recover variables in their notebook sessions.

Checkpoint and Restoration. Although not the focus of this paper, efficient kernel session checkpoint and restoration techniques are used in our system. Recent advances [8, 27, 28, 32] enable accurate and efficient dirty data identification for data science systems. The main idea is to identify dirty data by extending the standard serialization protocols without relying on centralized buffer pools. With this technique, Kishu [27] supports efficient incremental checkpoint and checkout for Python kernel session. The approaches can seamlessly integrate with existing notebook systems without requiring any intrusive changes.

9 FUTURE WORK

Future work should address the following research questions.

How to Support Collaboration of Exploratory Data Science? Kishuboard is mainly designed for individual use; however, data scientists often synchronously edit the same notebook when collaborating with others [24, 38, 50]. Some tools already support code collaboration [45] or online conversation when developing the same notebook together [51]. However, merging variables from different users remains an open problem. This new workflow would simplify many use cases like trying a variable training from one branch in another branch.

How to Support Semantic Search of History Commit? Currently, Kishuboard only supports searching previous commits by their attributes. However, sometimes the users would like to search semantically. For example, in a user study done by [23], users have queries with deeper semantics like "what was the state of my notebook the last time that my plot had a gaussian is the peak?" Some work [25] leverages advanced machine learning models to enable natural language search for cell code; however, it is unclear how to adapt the technique for semantics in both code and data.

How to Compare Alternatives at Scale? Comparison between alternatives is more challenging when facing a large number of choices. Currently, Kishuboard only supports diff between the two versions. However, [30] that their participants need to tackle many alternatives, ranging from hundreds of alternative graphs to thousands of data alternatives. Future works need to automate these comparisons through techniques like clustering or text mining.

10 CONCLUSION

In this work, we propose Kishuboard, a novel version control system for non-linear computational notebook versions where each contains both code and data states. Our goal is to help accelerate data science workflow while being intuitive to users. We interview industrial practitioners about suggested features that are useful. To achieve consistent code+data versioning while allowing users to check out code and data states separately, we formally define a consistency and build-in safeguard that allows safe checkouts. To simplify and visualize the 2D code and data states, we put forward a "git-tree-like" 1D commit history graph that is equivalent to the original 2D states. Kishuboard implements methods to help users easily locate relevant commits, including automatic folding, commit searching, and commit diff. We

demonstrate that Kishuboard can improve users' efficiency in many exploratory data science tasks and that the UI designs are sound through a user study with 20 participants. Our qualitative feedback also shows that 95% of the users are willing to use Kishuboard in their future work.

REFERENCES

[1] 2023. *Nbdime github repository.* https://github.com/jupyter/nbdime

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[3] Mode Analytics. [n.d.]. Modern Business Intelligence. https://mode.com//.

[4] Apache Arrow. 2024. PyArrow - Apache Arrow Python bindings. https://arrow.apache.org/docs/python/index.html.

[5] Benjamin Baumer and Dana Udwin. 2015. R markdown. *Wiley Interdisciplinary Reviews: Computational Statistics* 7, 3 (2015), 167–177.

[6] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? Pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–12.

[7] Supawit Chockchowwat, Zhaoheng Li, and Yongjoo Park. 2023. Transactional python for durable machine learning: Vision, challenges, and feasibility. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*. 1–5.

[8] Supawit Chockchowwat, Zhaoheng Li, and Yongjoo Park. 2023. Transactional Python for Durable Machine Learning: Vision, Challenges, and Feasibility. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning* (Seattle, WA, USA) *(DEEM '23)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3595360.3595855

[9] Taijara Loiola De Santana, Paulo Anselmo da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–34.

[10] Git SCM. 2024. Git GUI Clients. https://git-scm.com/downloads/guis A list of graphical user interface (GUI) clients for Git.

[11] GitKraken. 2024. Best Git GUI Clients for 2024. https://www.gitkraken.com/blog/best-git-gui-client An article comparing the best Git GUI clients available in 2024.

[12] Google. [n.d.]. Google Colab. https://colab.research.google.com/.

[13] Phillip Guo. 2013. *Data Science Workflow: Overview and Challenges.* https://cacm.acm.org/blogcacm/data-science-workflow-overview-and-challenges/

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[15] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[16] Fred Hohman, Kanit Wongsuphasawat, Mary Beth Kery, and Kayur Patel. 2020. Understanding and visualizing data iteration in machine learning. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–13.

[17] HuggingFace. 2024. HuggingFace - Pipelines. https://huggingface.co/docs/transformers/en/main_classes/pipelines. (Date accessed: July 10 2024).

[18] IPython Development Team. 2024. IPython Configuration Callbacks. https://ipython.readthedocs.io/en/stable/config/callbacks.html Accessed: 2024-09-10.

[19] Project Jupyter. 2023. Jupyter Notebook. https://jupyter.org/.

[20] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*, Vol. 10. 3025453–3025626.

[21] Mary Beth Kery, Bonnie E John, Patrick O'Flaherty, Amber Horvath, and Brad A Myers. 2019. Towards effective foraging by data scientists to find past analysis choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–13.

[22] Mary Beth Kery and Brad A Myers. 2018. Interactions for untangling messy history in a computational notebook. In *2018 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, 147–155.

[23] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–11.

[24] Catherine Li, Talie Massachi, Jordan Eschler, and Jeff Huang. 2023. Understanding the Needs of Enterprise Users in Collaborative Python Notebooks: This paper examines enterprise user needs in collaborative Python notebooks through a dyadic interview study. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–7.

[25] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. 2021. Nbsearch: Semantic search and visual exploration of computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[26] Zhaoheng Li, Supawit Chockchowwat, Hanxi Fang, Ribhav Sahu, Sumay Thakurdesai, Kantanat Pridaphatrakun, and Yongjoo Park. 2024. Demonstration of ElasticNotebook: Migrating Live Computational Notebook States. In *Proceedings of the 2024 International Conference on Management of Data*. ACM.

[27] Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, and Yongjoo Park. 2024. Kishu: Time-Traveling for Computational Notebooks. arXiv:2406.13856 [cs.DB] https://arxiv.org/abs/2406.13856

[28] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks. *Proc. VLDB Endow.* 17, 2 (oct 2023), 119–133. https://doi.org/10.14778/3626292.3626296

[29] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks. *arXiv preprint arXiv:2309.11083* (2023).

[30] Jiali Liu, Nadia Boukhelifa, and James R Eagan. 2019. Understanding the role of alternatives in data analysis practices. *IEEE transactions on visualization and computer graphics* 26, 1 (2019), 66–76.

[31] Jon Loeliger and Matthew McCullough. 2012. *Version Control with Git: Powerful tools and techniques for collaborative software development.* O'Reilly Media, Inc.

[32] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: multiversion replay with ordered checkpoints. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1297–1310. https://doi.org/10.14778/3514061.3514075

[33] Wes McKinney. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[34] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[35] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).* 561–577.

[36] Paperspace. 2024. NVIDIA H100 for AI & ML Workloads | Cloud GPU Platform. https://www.paperspace.com/.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[38] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. 2022. Eliciting best practices for collaboration with computational notebooks. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW1 (2022), 1–41.

[39] Deepthi Raghunandan, Aayushi Roy, Shenzhi Shi, Niklas Elmqvist, and Leilani Battle. 2023. Code code evolution: Understanding how people change data science notebooks over time. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* 1–12.

[40] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. 2023. Visualising data science workflows to support third-party notebook comprehension: an empirical study. *Empirical Software Engineering* 28, 3 (2023), 58.

[41] Lars Reimann and Günter Kniesel-Wünsche. 2023. An Alternative to Cells for Selective Execution of Data Science Pipelines. *arXiv preprint arXiv:2302.14556* (2023).

[42] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. , e1007007 pages.

[43] Russell Sears and Eric Brewer. 2009. Segment-based recovery: write-ahead logging revisited. *Proceedings of the VLDB Endowment* 2, 1 (2009), 490–501.

[44] Jeremy Singer. 2020. Notes on notebooks: Is Jupyter the bringer of jollity?. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 180–186.

[45] Micah J Smith, Jürgen Cito, and Kalyan Veeramachaneni. 2021. Meeting in the notebook: a notebook-based environment for micro-submissions in data science collaborations. *arXiv preprint arXiv:2103.15787* (2021).

[46] Sublime HQ Pty Ltd. 2024. *Sublime Merge.* https://www.sublimemerge.com/ Version control software for Git repositories.

[47] Krishna Subramanian, Ilya Zubarev, Simon Völker, and Jan Borchers. 2019. Supporting data workers to perform exploratory programming. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems.* 1–6.

[48] Satish Venkatesan. 2022. *Automatic Restoration and Management of Computational Notebooks.* Ph.D. Dissertation. Virginia Tech.

[49] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the loop: Supporting data comparison in exploratory data analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems.* 1–10.

[50] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–30.

[51] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. 2020. Callisto: Capturing the" Why" by Connecting Conversations with Computational Narratives. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* 1–13.

[52] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and restoring reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 138–149.

[53] Zijie J Wang, Katie Dai, and W Keith Edwards. 2022. Stickyland: Breaking the linear presentation of computational notebooks. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts.* 1–7.

[54] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. 2021. Fork it: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–12.

[55] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* USENIX Association, 15–28.

A ADDITIONAL FEATURES OF KISHUBOARD

A.1 Automatic Folding

Kishuboard naturally produces as many commits as the number of cell executions, which is typically in the order of hundreds, if not thousands across different sessions. Displaying all existing commits at once not only obstructs browsing and searching for a commit but also obscures the bigger picture to understand past works.

Kishuboard employs an auto-folding algorithm to visualize a large number of commits in the commit history graph, according to S4 from Table 2. As in Fig 14, Kishuboard folds a group of commits into one **grouped commit** on the graph and lets the user expand it to see the group of commits by clicking the group node (i.e., the rectangle with a plus sign).

Auto folding folds all commits automatically except for two types of commits. (1) **User-important commits**: auto folding does not fold the commits with user-defined tags or user-defined commit messages. (2) **Topology-important commits**: auto folding does not fold commits which are leaf (having no child commit), branch (having multiple child commits), root (having no parent commit), or merged commit (having two parents). Fig 14 illustrates commits that are automatically folded and those that are not folded with their reasons.

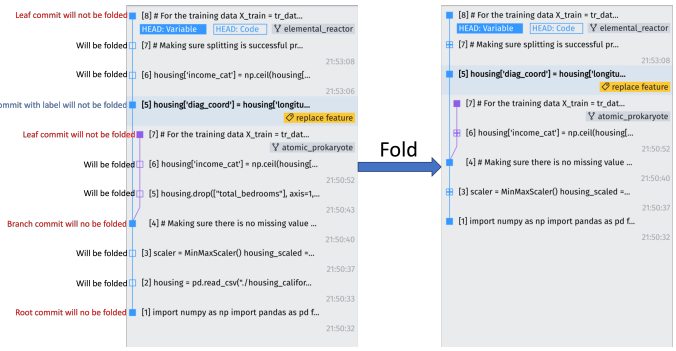
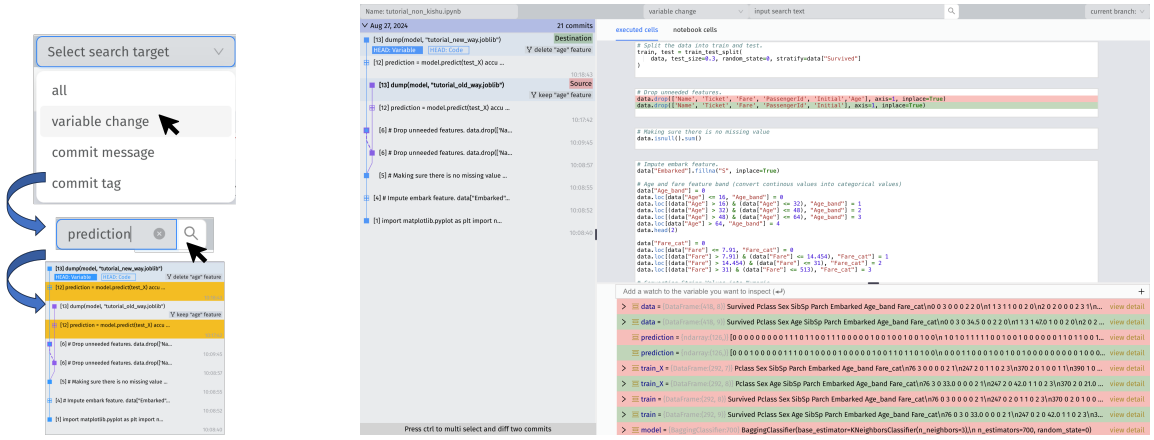


Fig. 14. Demonstration of how auto folding works in commit history graph. Other than commits with topology importance (e.g., leaf commit) as indicated in red annotations and user importance (e.g., tag) as indicated in blue annotations, all the other commits can be folded into grouped commits.

A.2 Commit Searching

Kishuboard's commit searching helps the user find commits based on their attribute. Like other search engines for Git, users can search for commits that have similar tag names and branch names and/or commit messages. Given a search query, Kishuboard finds and highlights all matching commits in the commit history graph.

Beyond conventional searches, Kishuboard also supports *searches by variable changes* (S5 from Table 2). Given a variable name, Kishuboard finds all the commits that change the variable value. For example, a user, Alice, may wish to understand how a training dataset `train_df` has been preprocessed. Through Kishuboard, Alice can search by `train_df` variable changes and find out about different variants of executed code cells that load the dataset, fill in missing values, and normalize the dataset across different experimental branches.



(a) Search commits

(b) Diff any two commits

Fig. 15. (a) Kishuboard's commit searching helps users find commits based on their attributes. Users can search for commits according to the commit message and tag. They can also search to highlight all the commits that change a specific variable's value. The figure shows an example of searching all the commits that changed the *"prediction"* variable's value. (b) Kishuboard's diff feature supports differentiating both code and variables of any two commits.

A.3 Commit Diff

Kishuboard's commit diff compares the differences between two commits not only in terms of code but also data states. Recall that, unlike source code files, a notebook contains a sequence of cells, which are strings of code. To compare code states between two commits, we adapt the existing method specialized for notebooks [1], which detects both across-cell differences (e.g., added, deleted, or modified cells) and within-cell differences (e.g., changed lines of code).

Meanwhile, Kishuboard maintains an auxiliary structure called **variable version table** that keeps track of the variable version for each commit. Each entry of a variable version table is a variable name and the latest commit that changes it. Fig 16a shows an example of a variable version table at commit V_3. Consequently, Kishuboard can quickly identify the difference based on the variable versions without reconstructing the variable values. For example, if a user compares the data state between V_3 and V_4, Kishuboard can lookup the two variable version tables and determine that `model` is different between the two commits, `fig` is deleted in V_4, and `app` is added in V_4.

Variable Name	Latest Change At
`data_df`	V_1
`model`	V_2
`fig`	V_3

(a) Variable Version Table at commit V_3.

Variable Name	Latest Change At
`data_df`	V_1
`model`	V_4
`app`	V_4

(b) Variable Version Table at commit V_4.

Fig. 16. Example variable version tables. Our system can quickly diff any two commits' variables by comparing their tables.