Talk: https://youtu.be/9HPhSBtgsXM

Fork It: Supporting Stateful Alternatives in Computational Notebooks

Nathaniel Weinman
nweinman@berkeley.edu
Microsoft Research/UC Berkeley

Titus Barik
titus.barik@microsoft.com
Microsoft Research

Steven M. Drucker
sdrucker@microsoft.com
Microsoft Research

Robert DeLine
rob.deline@microsoft.com
Microsoft Research

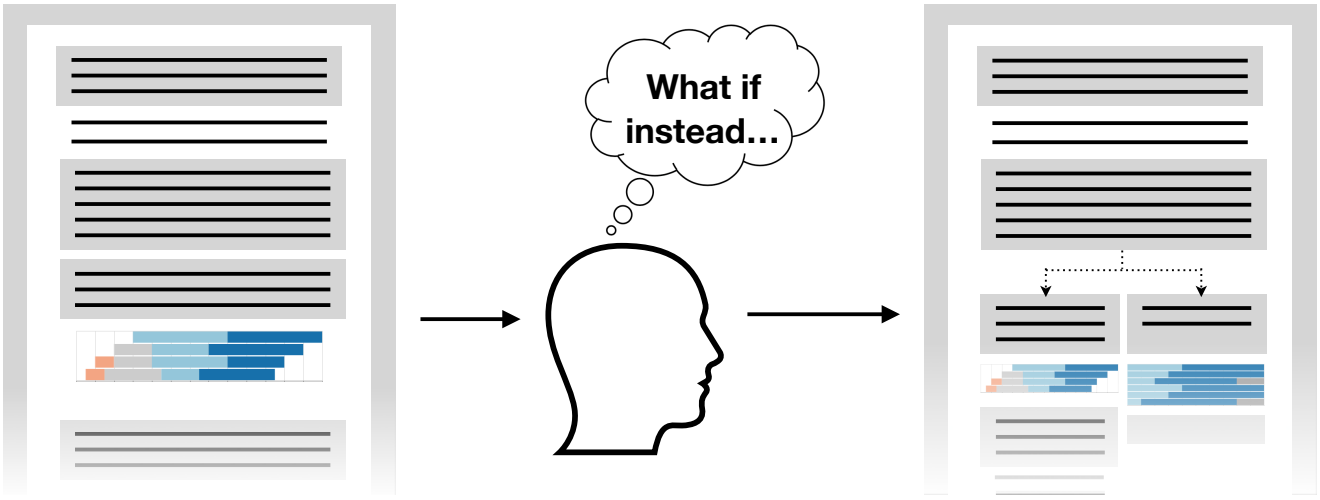


Figure 1: A user makes progress in a notebook (left) to test a hypothesis. They then wonder how an alternative approach may have worked out (middle). They fork their existing code in the middle of their notebook, starting their new exploration next to the old (right).

ABSTRACT

Computational notebooks, which seamlessly interleave code with results, have become a popular tool for data scientists due to the iterative nature of exploratory tasks. However, notebooks provide a single execution state for users to manipulate through creating and manipulating variables. When exploring alternatives, data scientists must carefully create many-step manipulations in visually distant cells.

We conducted formative interviews with 6 professional data scientists, motivating design principles behind exposing multiple states. We introduce forking — creating a new interpreter session — and backtracking — navigating through previous states. We implement these interactions as an extension to notebooks that help data scientists more directly express and navigate through decision points in a single notebook. In a qualitative evaluation, 11 professional data scientists found the tool would be useful for exploring alternatives and debugging code to create a predictive model. Their insights highlight further challenges to scaling this functionality.

Two new features

Backtracking is combined with forking

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445527

CCS CONCEPTS

• **Software and its engineering** → **Development frameworks and environments**.

KEYWORDS

Alternatives, computational notebooks, exploratory programming, code history

ACM Reference Format:
Nathaniel Weinman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2021. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3411764.3445527

1 INTRODUCTION

Data scientists engage in exploratory practices to understand data, to form and test hypotheses and to build predictive models [2, 6, 13]. Computational notebooks, like Jupyter, Databricks or Colab, are popular tools to support this work [21]. Notebooks present a flexible scripting environment for exploratory programming by providing the user with an arbitrary number of small editors, called cells, whose contents are submitted to an ongoing interpreter session. These cells can freely be visually reordered. The textual or visual results of a cell's code are presented directly below the cell. This interleaving of scripting code, visual results, as well as documentation in markdown, create a form of literate programming [15] that helps data scientists explain their work.

However, this same flexibility often leads to "messy" notebooks [9, 12]. Because cells are the only way to work with the interpreter, a data scientist will clutter the notebook with many cells of short-term value, for example to explore how a library function is called or to construct a temporary view of the data. At a larger scale, data scientists also explore many different decisions when working with their data [16], like different feature transformations, model attributes, or experimental design.

Exploring these longer-term alternatives often involves imperatively updating the interpreter's state. For example, to train a predictor, a data scientist might remove the outcome column from a table to ensure that it plays no part in training. Once the column is removed, the only way to recover it is to execute previous cells to reload the table. The most popular scripting languages, like Python, are also imperative. Exploring alternatives often involves assigning new values to variables. Whatever value the variable previously held is lost and can only be recovered by rerunning the code that generated the value (if that code still exists in the notebook). In contrast to previous research focused on the messiness of flexible cell executions [9, 12, 23], we address the messiness of polluting a single shared interpreter session.

We introduce a tool supporting two new interactions, *forking* and *backtracking*, that provide the user with multiple interpreter sessions in a notebook. Our tool allows the user to *fork* a new interpreter session, either from the current execution state or any previous execution state. That is, the user can either plan ahead to explore alternatives or can retroactively discover the need (which we call *backtracking*). The user interface presents these alternatives side by side, to allow ready comparison of their results. In a set of formative interviews we conducted with six professional data scientists, the most frequent need they described is better support for exploring alternatives. In a qualitative evaluation with 11 professional data scientists, we found that forking largely supports their needs and elicited additional requirements for the tool. Our contributions are as follows:

- The design and implementation of a tool providing data scientists access to multiple programming states within a single computational notebook.
- A usability study to provide insights into the uses and usability of managing multiple programming states.

2 RELATED WORK

2.1 Exploration in Notebooks

Data scientists explore a diverse set of hypotheses, theories, data, methods, and visualizations in their tasks, an example of Pirolli and Card's sensemaking framework [20]. Kery and Myers [13] observe how data scientists and other programmers engage in exploratory programming, or "open-ended tasks where a program's behavior cannot be specified in advance." They propose a framework for how programmers engage in this practice, finding many gaps in current tool support around sensemaking. Liu et al. [16] focus on the many decision points data scientists make to explore or revisit certain alternatives. They propose a framework for the triggers as well as the barriers to data scientists exploring alternatives given current tools, identifying opportunities for tools to better support this crucial behavior. When communicating the process of an exploration, the linear view of notebooks does not naturally lend itself to implicitly convey the explored decision points [14]. Interview studies have further found that these decision points are often not even explicitly documented [1, 10].

Notebooks are popular because the combination of iterative scripting and inlined visualizations supports sensemaking. However, this same support for iteration can cause challenges when data scientists want to explore alternatives. Wilson et al. [26] observe that one of the biggest challenges data scientists face is the inability to revert code. They propose notebook users leverage version control systems to address this challenge, though Kery et al. [12] find that formal version-control systems are too heavy-weight for data scientists. Instead, data scientists find work-arounds. Kery et al. [11] find that, in traditional programming environments, data scientists keep old versions of their code in comments. This requires them to keep a mental map of which variations have been tried and how those variations map to individual code snippets. In a later study, Kery et al. [14] found data scientists exploring alternatives by replicating code across many cells and then later refactoring—another tedious and error-prone work-around.

Rule et al. [23] explore the tension between using notebooks for exploration (overwriting cells, exploring many alternatives) and explanation (providing a clean, carefully maintained narrative) by analyzing two large sets of notebooks and conducing interviews with academic data analysts. They highlight the importance for tools to better support organization without much additional effort to the data scientists. In particular, they note the limitations of the linear style of notebooks. As previous studies document the need for better support for exploring alternatives, we included a focus on these explorations in our formative interviews.

2.2 Support for Revisions in Notebooks

Because of the rapid iteration in exploratory data analysis, data scientists often want to return to previous versions of their work. Researchers have created various kinds of support for this need. Mikami et al. [17] present a micro-versioning tool focused on supporting experiments in exploratory programming. They explore how to provide an effective way for users to make sense of the code they are undoing, creating a hybrid of regional and tree-structured undo models. Janus [22, 23] allows users to hide cells and view previous versions of cells. Verdant [12] allows data scientists to

navigate through previous versions of a notebook, through Git-like functionality. Gather [9] supports users in producing the ordered, minimal code to reproduce chosen results in the notebook. While these tools focus on organizing and exploring the history of the *content* of the notebook, forking and backtracking focus on organizing and recovering the *execution state*.

2.3 Non-Linear Presentations

Researchers have developed a range of tools to support visual representations of code organization beyond traditional file structures. Hartmann et al. [8] presented Juxtapose, a tool with a runtime environment that executes partially linked code variants in parallel and displays their outputs in a grid. Bragdon et al. [3] presented Code Bubbles as a tool to improve code comprehension. It allows the user to use lightweight, editable fragments called bubbles to group and organize code fragments. Nelson et al. [19] extend this idea beyond code artifacts, proposing stackable cards as a helpful way of dynamically organizing artifacts at different levels of abstraction, such as code, design documents, task lists, email messages, and more. In Debugger Canvas [5], the user navigates through parallel call paths in a debugging session. The focus on stepping into parallel call paths begins to expose a tree-like structure to the user as they make sense of the execution of their code. We take inspiration from these previous non-linear presentations of code when we present forks side by side.

3 FORMATIVE INTERVIEWS

We interviewed 6 professional data scientists (F1–F6) at Microsoft, a large, data-driven software company. Participants had 2.5–15 years of experience. The interview lasted 60 minutes, for which informants were compensated $25. We structured the interview around several potential extensions to notebooks. The informants were most enthusiastic about tools that break the single linear (possibly out of order) model of notebooks. These semi-structured interviews yielded several key ideas that yielded design principles for our tool.

3.1 Current Pain Points

Data scientists regularly explore different hypotheses, or "alternative universes" [F3]. As notebooks do not directly support these alternatives, the informants had developed strategies to manage this mismatch between their conceptual process and the notebook model. However, they described these strategies as "laborious, gross" [F4] and "hodgepodge" [F2]. a confused mixture of different things

One strategy data scientists employ is being careful in organizing their notebooks and naming variables. However, F4 notes that one "only learns that discipline by being burned a few times." F2 rigorously documents their notebooks, trying to ensure that if they view it again a year later, they will be able to understand their logic. F4 uses markdown cells to distinguish different hypotheses, and "if they're going to work on hypothesis 1 again...they rerun all of hypothesis 1" to ensure all variable values are correct, even if the computation is time-intensive. F6 carefully names their variables, ending up with "either uninformative names like `df_1` and `df_2`, or increasingly onerous names like `df_no_personality`". Even with this strategy, however, they often "forget about the names and then have to go up and reread how they defined it." Though these informants experienced the same limitations of the notebook, they came up with distinct coping strategies.

Even with careful organization, understanding what differs between two hypotheses is difficult without careful documentation. F4 uses markdown cells to distinguish different hypotheses, and they'll "be subtly different from other hypotheses but good luck finding where". F5 "ends up having clones of so many notebooks of different approaches where the only diff on them is the last 2–10 cells", which they then have to carefully refactor to keep intended parts consistent.

3.2 Expressing Alternatives

There are many stages to data science tasks, such as cleaning data, engineering features, training models, and evaluating models [2]. Data scientists often explore multiple alternatives for some of these stages, such as "trying a bunch of different models and parameters to find which one works right" [F5]. However, when trying different models, the work in previous stages, like cleaning data and engineering features, is shared among all of the models.

The informants underscored the importance of exploring alternatives along with current limitations. F3 notes that "the process just isn't linear. There's a logical branching of what you're doing where you're comparing two alternative universes." F6 reinforces the prominence of alternatives in their workflow, "where they have to try three different things in order to know what's the better choice". F5 echos this sentiment, saying data scientists "try a bunch of different models and parameters and find which one works right." F4 highlights a particular challenge of current notebooks, that when a particular approach "is bad, which happens often, they'd like to delete it and go to what they had before without" having to "run many cells again" as a type of "undo magic for the three different cells" which explored the now debunked approach. In short, informants expressed the need for a direct way to explore alternatives. We use the term *fork* for this, based both on the common idiom of "a fork in the road" and more technically the Unix concept of a process fork, which replicates execution state. We avoid *branch*, since many data scientists use branches in Git, which is both more complex (e.g., merging) and does not involve execution state.

Underscore: underline (something); *sublinhar*

3.3 Execution History

Data scientists iterate with their data, often using the results of some analysis to decide on relevant alternatives to try [2, 18]. It is not uncommon for data scientists to analyze the false positives and negatives of a model to determine the next model they might try. For example, F2 recalls times when they thought "they didn't need some columns and then [several cells later] realized that they do" for a new hypothesis. In current notebooks, data scientists sometimes try to organize their notebook so that it will work as expected if cells are run in a fresh kernel from top to bottom. However, it is challenging to keep notebooks in such a clean state [9].

In the context of forks, this suggests that data scientists would benefit from being able to "backtrack", or move back up the tree, to create new forks and explore alternatives such as modifying a

cell and rerunning the cells below it. The data scientists we interviewed further expanded on how they might use this functionality to address any type of regrettable code. F3 thinks of it "like a time machine," returning to "how it was supposed to be before they...messed it up." F5 often "wants to go back a couple cells because they borked the data in some way that is difficult to revert". F5 uses git to "check in their code constantly" precisely to go back to previous code. Some tools, such as Databricks [1], partially address this by automatically checkpointing the visual layout of notebooks to allow users to return to previous versions. However, users must rerun all relevant cells as it does not save execution state—an impossible task, if critical code cells have been overwritten or deleted. Allowing data scientists to return to previous states both fits their reported work style and overcomes difficulties due to overwritten or deleted cells.

3.4 Visualizing Alternatives

Forks provide independent execution states on different paths. How should a fork be represented in the notebook user interface, especially since the existing layout (in particular, cell ordering) does not reflect the execution state? Two informants lamented at how it is "a big pain to be constantly scrolling back and forth" [F3] between competing hypotheses, as notebooks layout all cells in a single column. F1 notes that they wish they "were able to do their visualizations side by side rather than having to them in individual cells" to more easily compare them. F6 encourages a side-by-side view, as "visually you know it's two different paths". In short, we chose a side-by-side layout for fork paths.

4 EXAMPLE USAGE SCENARIO

We describe a short scenario to convey the experience of using our forking and backtracking tool in a Jupyter Notebook. Lily is a data scientist who uses Jupyter Notebooks in Python. She is performing exploratory data analysis to build a model that can predict song genres, based on a large data set. This section shows how this tool helps her organize, explore, and communicate her findings during data analysis.

Lily begins by importing relevant libraries and loading her data set. After loading the data, she begins her exploration. She plots distributions of columns, explores correlations, and investigates missing values. Several cells and some expensive computation later, she is now satisfied with her preliminary understanding of the data and is ready to start generating some features she thinks will be useful.

4.1 Containing Messes

Lily notices that the Track Release Date is part of the data set. This seems like a promising predictive column, as different song genres have been popular at different times. She wants to transform this column from a string into the days since Jan 1, 1950. Unfortunately, when browsing the date values, she notices that there are a few different formats in this column. Some appear to be in `MM-DD-YYYY`, some in `Month D, Yr`, some simply have the year, and more variations. She knows she has to process all of these different formats, but is also a little rusty with `datetime` [2] conversions in Pandas.

Lily approaches tasks like this by iteratively writing code for each format one at a time until all have been addressed. But, she also wants her final notebook to have a single function that fully handles all cases in this column to make it easier for others to read. Lily creates a *fork* below the current cell, giving her two side-by-side paths to work with. She will use the right *path* to experiment across cells as she discovers the correct code and the other path to write the final conversion function.

In both cases, Lily knows she'll want to modify the dataframe directly, not make a copy of it in a new variable. On the right, she experiments writing code to handle dates first in the MM-DD-YYYY format. After a few tries, she gets it working and verifies it by checking some rows on the dataframe. Next, on the left path, she writes a new function which conditionally applies the code that she just created on the left path (Figure 2). Back on the right, she now tries to tackle the next case in a new cell. Again, after getting it to work and checking some specific rows, she updates her function on the left. After repeating this a few times, she has successfully transformed all the date formats in the dataset.

Finally, Lily runs her new functions on the left path, then plots the distributions of her new columns. Seeing that they look identical, she is convinced that the conversion function accurately represents the work she did in multiple cells in the other path. Because each fork uses its own execution state, she can be sure that her conversion function works on the original dataset and has not been influenced by her exploratory work in the right path. With this task accomplished, she deletes the right exploratory path. Lily carries on with her work, her dates properly converted and her notebook free of the short-term explorations needed to remember the `datetime` functions.

4.2 Comparing Approaches

Lily continues to transform and clean the data. Though she has been diligent in ensuring her notebook can run from top to bottom, it still takes quite a bit of time due to all the processing. She continues to train and tune a random forest model. However, no matter how she tunes the parameters, she cannot quite get the accuracy she wants. She decides to see if a neural net model might work better.

Lily uses the `backtrack` dialog to browse the last few cell executions to get to the execution state she would use for training the ~~the~~ neural net (Figure 3). In this case, it is a cell where she had transformed Track Tempo from a numeric variable to a categorical variable distinguishing fast and slow songs. Lily had previously transformed this column as part of training a random forest. She has already overwritten the original numeric column from her local variables, and it would take some time to re-run all the cells above. However, by backtracking, she is able to move back to the state just before she overwrote the column.

Selecting a state in the backtracking dialog creates a fork. On the fork's left path are all of the cells for training the random forest, that is, those cells executed after the chosen state. On the right path is an empty cell to allow Lily to start an alternative exploration.

[1] https://databricks.com/

[2] https://docs.python.org/3/library/datetime.html

Figure 2: An example of two forked paths. The cell on the left is being used to slowly construct a clean function that will be part of the final notebook. The cells on the right are being used for less cautious exploration as Lily determines the correct code.

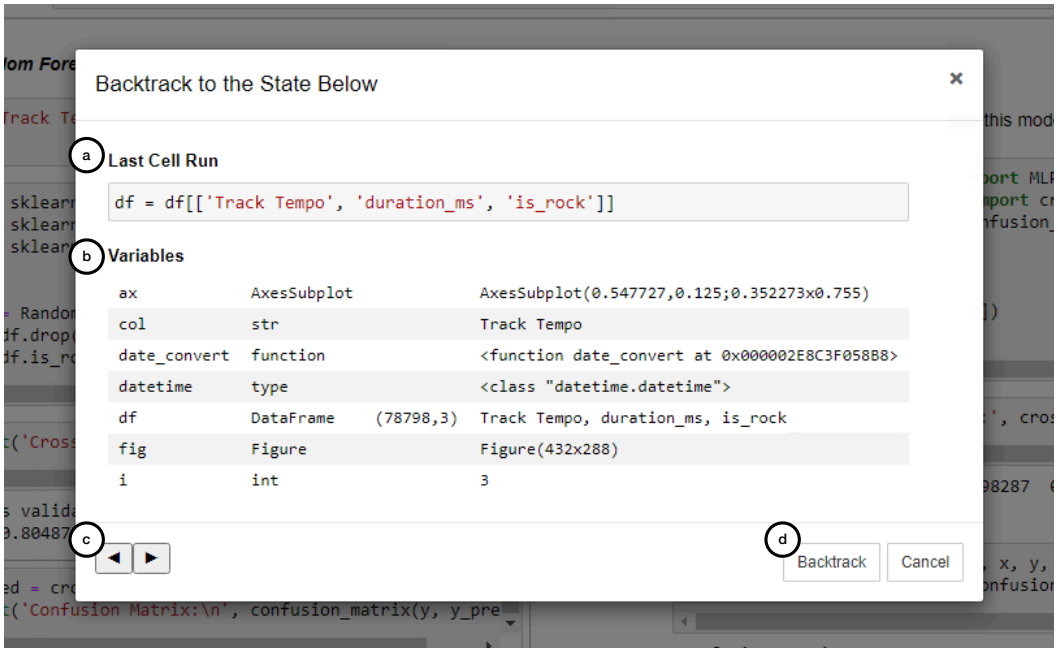


Figure 3: The backtrack dialog, to aid users in navigating the the appropriate previous state. (a) The code from the last run cell at the current point in history. (b) The values or summary statistics of all variables at the current point in history. (c) Navigation arrows to move between different points of history, each click goes to the last or next executed code cell. (d) A button to confirm that a new fork or path should be created which is backtracked to this state in history.

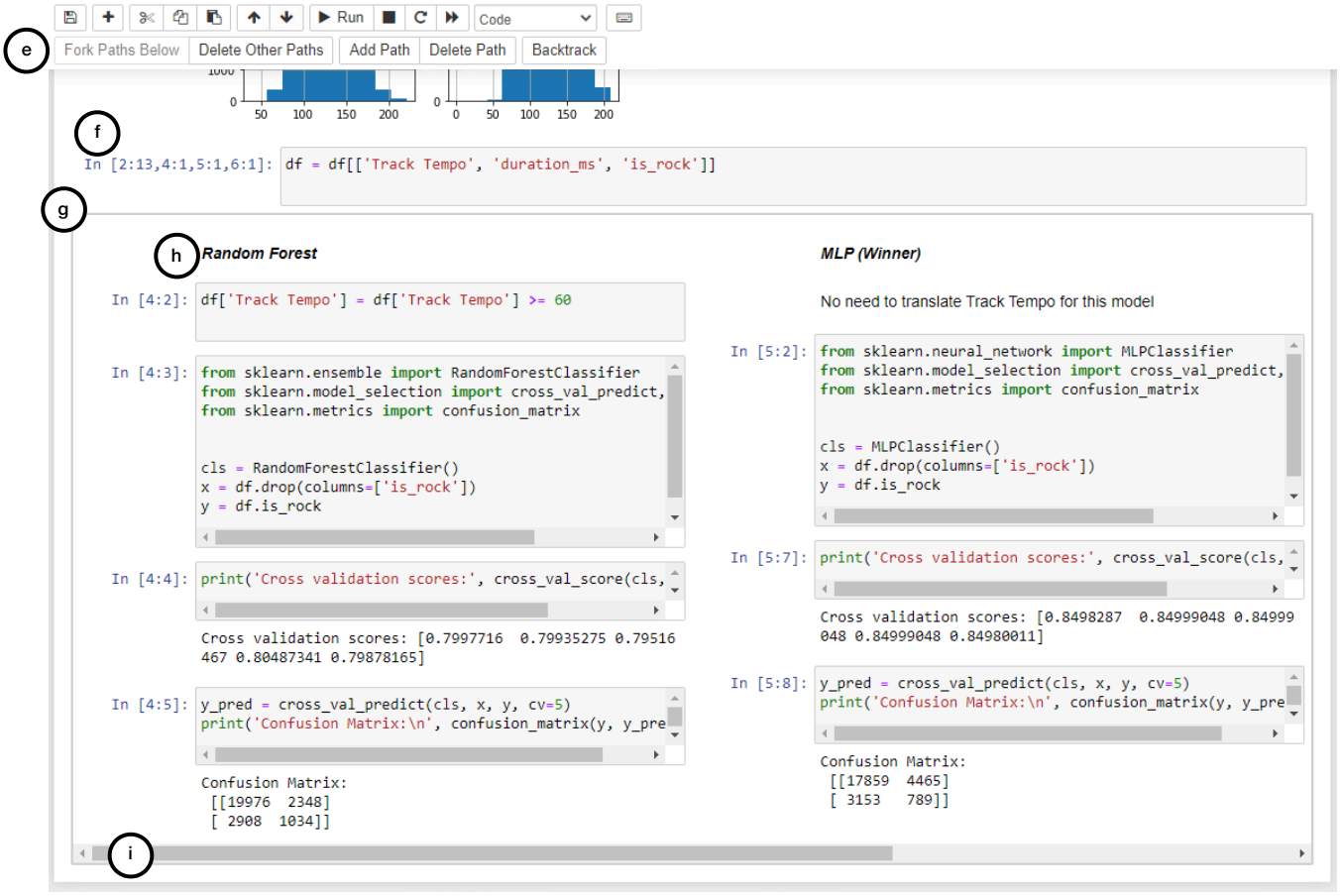


Figure 4: A view of the forking and backtracking tool. (e) a series of navigation buttons related to forking and backtracking. (f) An example of the modified run counters for a cell that has been executed on 4 kernels. (g) A bounding box representing the start of a fork with multiple paths. (h) The title markdown cell for a path encouraging the user to describe it. (i) A horizontal scroll bar, present when there are too many paths to fit on the screen (in this case there are 3 paths).

Some of the cells from the previous alternative are still relevant, so she copies them from the left path to the right. She then writes new code to train and ~~and~~ tune her neural net. Once the neural net is working, she compares results on the left and right paths. In addition to accuracy, she decides to compare F1 scores. She writes the code to compute this metric, first on the left path, then on the right—both paths have active executions. By comparing the new metrics, she decides the neural net is best for this task.

4.3 Communicating Decisions

After getting the above models working, Lily similarly explores a K-nearest neighbor model, but is unable to outperform the neural net. Though Lily has made her choice for the best performing model, she still wants to share the alternatives that she tried. She keeps this fork in her notebook, with clearly labeled titles indicating which she found to be the better choice. When Lily shows her notebook to a peer, they can clearly see the initial exploration and pre-processing she did, followed by the decision point to use these three models for prediction, and finally the relevant code for each of those paths (Figure 4). This allows Lily's peer to understand where Lily made her decision, with a side-by-side comparison of performance metrics. Steps that are needed only for one alternative, like the transformation of Track Tempo for the decision tree, are shown on only the relevant path.

5 IMPLEMENTATION

Our tool is built on Jupyter Notebooks [24], a popular open-source computational notebook[3] (Figure 4). Our tool provides users access to multiple executions states through forks, points where an existing execution state is copied into new, independent Python kernels. Forking points (g) are visually represented with multiple side-by-side sets of cells in a bounding box. Each of these sets of cells represent a different path. Each path begins with a markdown cell (h) with an auto-generated title to help users keep track of the intent of each path. Paths are limited to 50% of the width of the

[3]https://github.com/jupyter/notebook

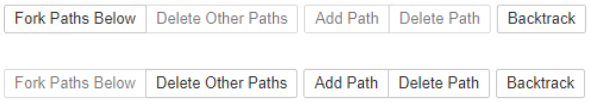


Figure 5: A close-up of forking and backtracking navigation buttons. The top row of buttons represents what the user sees in "normal" mode, when there is no fork. The bottom row of buttons represents what the user sees in "forking" mode, when there is a fork. Note that all the buttons, with the exception of Backtrack, toggle between these two modes.

https://github.com/uqfoundation/dill

screen to prevent them from becoming too narrow. The interface introduces a horizontal scroll bar (i) if necessary to navigate between 3 or more paths.

There are two ways for users to create new forks. Firstly, they can *proactively* create a fork through the *Fork Paths Below* button (e). This moves all the existing cells, if any, into the left path. Using dill[4], a robust library for serializing and de-serializing Python's execution state, the tool saves the current execution state. It then creates two new Python kernels, one for each path, and loads this state into each of those kernels.

Alternatively, users can retroactively create a fork through the *Backtrack* button (e). As seen in Figure 3, a modal dialog appears that allows users to browse the history of every cell execution (c). At each point in history, users can view the executed cell's code (a) and a summary of all defined variables (b). This helps users understand the execution state to which they are navigating. Once finding the right point in history, users can click *Backtrack* (d) to create a new fork from that historic execution state. Visually, the fork is created just above the highest cell which was undone. Our prototype uses dill to save Python's execution state after every cell execution, though we discuss below how this could be made more efficient.

After a fork is created, the user is now in "forking" mode and cannot create any new forks. As seen in Figure 5, buttons are enabled and disabled depending on the mode. To keep the interface simple, our tool only supports one fork at a time, rather than nested forks. However, users can have an arbitrary number of paths on a fork. Users can create a new path with the *Add Path* button, which creates a new empty path from the same execution state that was saved when creating the fork. Users can also create new paths with the *Backtrack* button, moving the fork point up if necessary. Users can delete individual paths, e.g. if they are no longer relevant, with the *Delete Path* button. Alternatively, if there is a "winning path," such as selecting the best performing model, users can *Delete Other Paths*. This will keep only the cells in the selected path and will return the notebook to "normal" mode, allowing for future forks.

5.1 Managing Kernels

Our representation of forks, even when limiting users to a single fork, presents interesting challenges for interpreting certain execution commands. Consider a user that has done some pre-processing of the data, then forks to create two models. They realized they forgot to drop a column. Taking advantage of notebooks out-of-order execution model, they add a cell before the fork to drop that column.

In this case, our tool will execute that code on all active kernels, clearly visualized to the user in an unobstructive way (f). Instead of having a single run count for a cell, all run counts are prefixed with a number representing the kernel or kernels the code was executed on. Additionally, when forks are created, the original kernel, or the "above-fork" kernel, is maintained in addition to the new ones representing each path. Maintaining this "above-fork" kernel is slightly less efficient, but smoothly supports the case where users would update code above the fork, such as pre-processing code, and then rerun all the cells within a path, such as transforming the data uniquely to the target model. Due to time constraints in developing the prototype, only the output from the "above-fork" kernel is shown. How to ensure consistency

Additionally, some of the interviewed data scientists try to keep their notebook organized by occasionally using "Restart & Run All", which resets the execution state and runs the cells sequentially from top to bottom and ensuring the results do not change. This is a common technique to ensure notebooks are reusable. To support similar functionality, several hidden cells are added when forks are created. Just before each fork, a cell saves the current state of the "above-fork" kernel to a file. At the start of each path, a cell loads that file. This way, any non-deterministic functionality in the code before a fork, such as sampling rows of a data set, remains consistent between all paths if a user selects "Restart & Run All" or similar commands.

https://github.com/microsoft/AMBROSIA

5.2 Future performance optimizations

As one participant noted, they're often working with data so large that they "can't fit even 3 copies of the data in RAM". There is a major limitation of our current tool around scalability, as it saves slightly compressed representations of state for every single cell execution. For the purposes of our study, this was not an issue, as our data sets were sufficiently small. However, this limitation would have to be addressed for actual deployment.

Though we did not address it in our prototype, we note several ways to optimize the tool. First, the tool could leverage Ambrosia [7], which provides "virtual resiliency," namely, distributed, performant support that could be used to externally save and load varying checkpoints of environment state. Alternatively, we could optimize this with techniques such as program slicing [25], a static analysis technique which identifies which lines of code contribute to the current value of a variable. Program slicing could be used to intelligently analyze which variables were modified to store only modified variables instead of entire execution states from each cell execution. Similarly, for structured data, like tables, we could store only differences in state rather than entire copies. Of course, such optimizations are not worth the effort until the tool has been proven to be valuable to users.

6 QUALITATIVE EVALUATION

We designed a usability study to explore how participants leverage forking and backtracking in an exploratory predictive task. The study was designed to answer the following research questions:

[4]https://pypi.org/project/dill/

- How do participants use forking in exploratory data tasks? (Section 7.2)
- How do participants use backtracking in exploratory data tasks? (Section 7.3)
- What are the key challenges to scaling to more complex tasks? (Section 7.4)
- How effective is the tool in supporting the original design motivations? (Section 7.5)

We invited 200 randomly selected data scientists at Microsoft, a large, data-driven software company. Participants were required to have at least one year of familiarity with Jupyter Notebooks and to have familiarity with the sklearn[5], a popular Python library for machine learning. We recruited 11 participants (P1–P11, 23–46 years of age, 3–28 years of experience). Seven participants reported notebook use every day, one every other day, two once a week, and one did not report. Participants were compensated $25/hour for their time.

Each 90-minute session began with the participant signing a consent form. The session then consisted of two tasks to build prediction models. For the two predictive tasks, we gave participants two existing datasets, one from Tidy Tuesday classifying songs by genre[6] and the other from Kaggle predicting IMDB movie scores[7]. We chose these datasets because they were of comparable total size, involve non-technical domains, and are complex enough to be worthy of analysis. We ~~conterbalanced~~ counterbalanced the use of the two datasets between subjects.

The remainder of the session consisted of four parts: (1) using a standard Jupyter Notebook to build a predictive model (15 min); (2) a tutorial on forking and backtracking (10–20 min); (3) using Jupyter Notebook with forking and backtracking to build a predictive model (15–25 min); and a final questionnaire. The questionnaire asked how well each of these experiences supported their tasks, as well as how they might apply forking and backtracking in their daily work. The purpose of using a standard Jupyter Notebook was to aid in comparing the experiences, rather than making quantitative comparisons. Hence these two experiences are not counterbalanced.

This study took place at a time when in-person studies were not feasible. The participants took part in this study on a single monitor through a remote desktop onto a virtual machine hosted on a cloud platform. They used their own machines with a variety of monitor sizes.

7 RESULTS

We analyze how participants use, or do not use, forking and backtracking in the usability study. We additionally analyze participant responses to survey questions as well as utterances during the study.

7.1 Effectiveness of Tool and Features

After completing all tasks in the study, participants were asked to fill out Likert-scale questions about how they would use this tool. One participant was unable to answer these questions due to a computer crash at the end of the study.

[5]https://scikit-learn.org/
[6]https://github.com/rfordatascience/tidytuesday/tree/master/data/2020/2020-01-21
[7]https://www.kaggle.com/carolzhangdc/imdb-5000-movie-dataset

Questionnaire based on the Creativity Support Index (CSI)
https://dl.acm.org/doi/10.1145/2617588

The first set of items asked how well each experience supported their tasks, on a 5-point Likert scale directly comparing the two: (1) standard notebooks were very much better; (2) standard notebooks were somewhat better; (3) the two were equal; (4) fork and backtrack were somewhat better; and (5) fork and backtrack were very much better. The items are based on Exploration, Results Worth Effort, Expressiveness, and Collaboration items from the Creativity Support Index [4], a psychometric survey designed to evaluate how well a tool assists a user in their creative work such as data exploratory analyses. As can be seen in Figure 6, participants found the tool useful across all of these dimensions, finding the tool particularly effective at supporting Exploration and Results Worth Effort.

The second set of questions asked participants to rate the helpfulness of certain tool functionality on a 3-point Likert scale, with an option to decline if they did not have enough information. As can be seen in Figure 7 participants found all features would be helpful to their everyday work, even if they did not use all those features in the study.

7.2 Forks

Data scientists make many decisions over the course of an exploratory task. By providing a clear interface that allows participants to express these decisions, we gain insight into some of these decision points. 9 of the 11 participants created forks over the course of the usability study, creating at most 3 (median=1) forks with at most 3 paths (median=2) at a time. P3 did not use the tool because they wanted to "very carefully" understand the functionality before using it, though still thought the tool would be useful to their everyday work. P10 had technical issues effectively cutting down their time on the task, but noted they would have created a fork after making more progress on the task.

7.2.1 Forking Opportunities. We find that participants often used forking to *compare* decisions. P7 used forks to compare alternative visualizations of the same columns to understand the relationships between columns in the data. P1 and P9 used a fork to compare different approaches for handling null values in the data, continuing to build the same model in both paths. 6 participants used forks to explore different subsets of features to feed into the models they built. Finally, P4 used a fork to compare multiple different classifier models.

However, participants also appropriated forking beyond comparing alternatives. Participants used forks to *organize* their code and contain messes as they worked. In particular, they dedicated a path as a safe "playground" for code that was dangerous or not to be included in their final "clean" notebook. Early on, P1 used a path to explore how transforming a categorical variable into dummy columns would affect the size of the data and subsequent performance of the task. Later in the study, when a particular model failed, P1 created a new path to explore the data and try to address the issue. P6 exhibited similar behavior. P9 echos this use, appreciating the "seamless transition from main code to "experiment" code...without creating duplicate code or extra if-else logic."

Additionally, participants used the side-by-side layout of forks for *parallelism*. P4 used two paths to rapidly iterate adding and removing different feature columns, maintaining the better-performing path at each iteration and then modifying the other

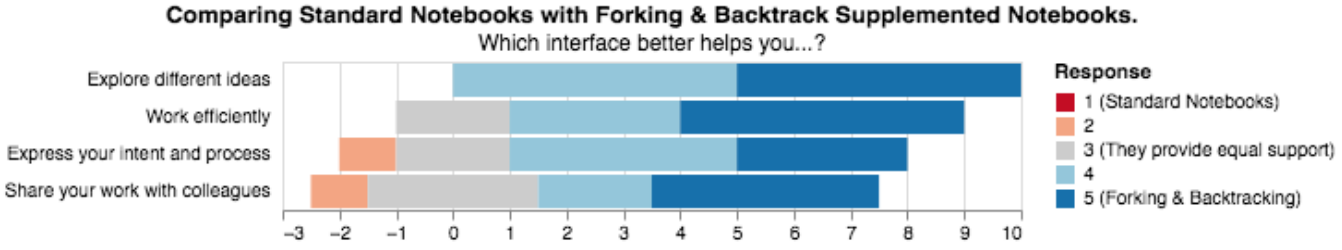


Figure 6: Results from Likert-item questions exploring which interface better supported certain goals. Participants found forking and backtracking functionality most helpful for exploration and least helpful for collaboration.

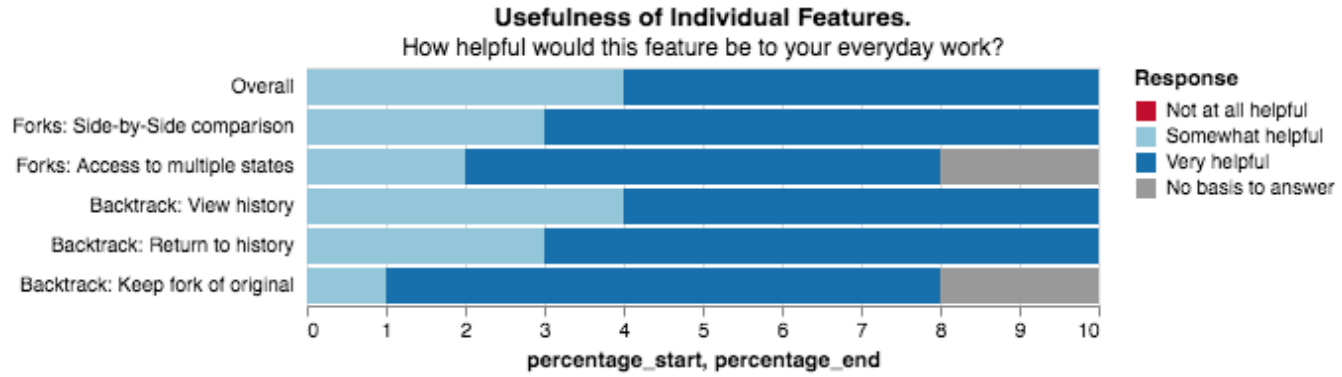


Figure 7: Results from Likert-item questions determining the helpfulness of various features of the tool. Participants thought all features would be helpful to their everyday work

for the next experiment. P11 took advantage of long-running tasks, such as hyper-parameter tuning, to context switch to another line of work and return as soon as the lengthy command had completed.

It worked, but the participants didn't understand how it works

7.2.2 Confusion of "Above-Fork" Execution. 8 of the 9 participants that created forks executed code on an "above-fork" cell, for example when importing additional libraries in their first cell. This clearly indicates a need to support out-of-order execution even when a fork is present. The execution behavior worked for 7 of them, allowing them to continue with their work without issues. When it did not work, a "Restart & Run All" was able to fix the issue.

However, despite this not interrupting their flow, several participants expressed their concern reflecting on the task that they did not have a good understanding of how or why it worked. The tool could address this by following a similar approach to Git or VarioLite [11], creating a new fork whenever users re-execute "above-fork" code. It would also be interesting to run a more targeted study to understand the expected or desired behavior of running "above-fork" cells to see if a less disruptive design could address this issue.

7.2.3 Sharing code across paths. A few participants noted the frustrations of copy-pasting similar cells between paths, e.g. the model building and testing code after selecting a different subset of features. One participant, P7, went as far as creating a list of function arguments in an "above-fork" cell and unpacking them in the duplicate function calls in each fork.

Jupyter notebooks currently support functionality to copy and paste cells. Though this would reduce the impact of this limitation, it is an infrequently used feature that many may not know about. Alternatively, the tool could be improved by supporting "linked" cells, similar to Juxtapose [8], which would contain the same code across multiple paths. This would allow users to express code common across multiple paths, for example, code to compute performance metrics for models on different paths.

7.2.4 Width. The most frequent critique, though easy to address, was the narrow width of cells in different paths. This could be partially addressed by simply "having the Jupyter notebook scale to 100%" instead of setting a maximum width as it currently does. Despite the complaints of narrow cells, however, participants still highly valued the ability for side-by-side comparisons. The tool could be greatly improved by providing more functionality for selecting visible paths. Simply allowing "only showing one fork at a time...and switching between them" [P4], in addition to the side-by-side comparisons, could give users sufficient control to address this concern.

7.3 Limited Use of Backtracking

2 participants used backtracking as they constructed their models. After P2 generated their first model, they used backtracking to undo four cells' executions and to try an alternative set of features. P5 used it to debug a data quality issue, after corrupting the data in a

Preemptive: made so as to deter an anticipated unpleasant situation

https://en.wiktionary.org/wiki/preemptive

way that would have been difficult to recover from. No participants used backtracking to select historic execution states for comparison.

We believe there are two reason why this feature did not see more use. First, 20 minutes may have been insufficient time for the problem to occur. As P11 stated, they "didn't use back tracking because they didn't have enough time to get to a point where they would have done this."

Second, some participants did not recognize the opportunities to use backtracking for comparisons. P6 and P8 both created forks retroactively, modifying cells that had already been written. However, they used the "Create Fork Below" functionality, which was designed for proactive forks, copying the current execution state. One possible explanation is the tutorial may have presented this feature too narrowly by focusing on correcting errors. Participants also have ingrained habits around statement management, for example, rerunning cells to re-create states by hand, which compete with the backtracking feature.

7.4 Scaling to More Complex Tasks

7.4.1 Visualizing State. 4 participants expressed concern in their understanding of which values of variables are present in each path. P4 wished "it was clear what variables and classes are available to which kernels". However, P10 noted that while "it can be a little confusing to understand the state of a cell that you are going to run at first...it isn't bad after using it for a few minutes." This concern could be addressed with more intelligent and persistent sidebar displays, for example showing the current summary of variables as done in the backtrack model.

7.4.2 Supporting Full Trees. For simplicity, our prototype only allows a single fork at a time, which the user interface enforces by maintaining separate "normal" and "forking" modes. This meant participants could only express a single fork point at a time, even if there could be an arbitrary number of paths. P6 wishes they could "archive paths (instead of deleting them)...for the option of reviving them later". P9 expressed a desire for "nested forks...when taking different approaches at different stages of an experiment." F3 from the formative study similarly expressed a desire to "keep them both to say, look, this is the performance of this one". This limitation might be more of a hindrance in the complexity of our participants' daily work. Future work could address this with helpful visualizations, such as a hierarchical organization in "a small side navigation like [folder structure in] VS code" [P3].

7.5 Evaluating our Design Principles

We revisit how well our tool supports our original design motivations, in the light of user feedback.

7.5.1 Expressing Alternatives. The implementation of using forks to express alternatives "makes perfect sense" [P5], "is awesome" [P10], and offers "easy exploration" [P6] "without having to worry about the state of the notebook" [P9]. Participants found forking to a be a natural metaphor to express the decision-making process already in their work, using it in a range of different ways. One participant in particular was grateful to finally have a way to "organize the age-old question of "have you tried doing it this way?"" [P11].

Though many participants found forking, in particular preemptive forking, to be a natural representation of their process, there were some concerns about using notebooks with forks as an artifact for communication. P1 liked that, in traditional notebooks, "you can follow the top to bottom train of thought". P8 mentioned they'd only use forking and backtracking "for exploration that I didn't need to share heavily with others". This is particularly surprising given how "intuitive" [F3] the idea of forking is. Regardless, forking clearly provides a good opportunity for data scientists to express alternatives for their own organization.

Forking seems more interesting for exploration, not communication (non-linear exploration vs. linear story)

7.5.2 Manipulating Execution. P11 was thrilled to "not have to rerun the entire notebook when they accidentally mess something up (which happens often)!". Most participants were thrilled to be able to easily reverse regrettable code. P6 noted that they wouldn't use this feature often, though, as they organize cells and variables such that, when they fix a mistake in a cell, they can simply run all the cells below. As P7 mentions, though, for many notebooks it is common to "have a lot of cells they need to run to get back to their preferred state," emphasizing the need for backtracking for many.

However, though participants were excited to undo cell executions, the benefit of forking as part of backtracking was less obvious. Participants did not seem to realize that backtracking could be used to simply to explore alternative paths from previous states in the code. We originally conceived of backtracking as a form of forking that does not require forethought. However, based on participant feedback, having a light-weight way to browse and resurrect previous execution states is so useful that it may be better to make this its own independent feature.

Based on the feedback, a simpler way to go back to previous execution states may be sufficient

7.5.3 Side-by-Side Presentation. This prototype took a step towards providing more expressive organization of cells and content in Jupyter notebooks. Before seeing our tool, P1 expressed a desire when working with notebooks to "branch one of the data processing tasks and compare them side by side". Despite complaints about the width of the window, participants expressed strong support for organizing different hypotheses and techniques adjacently. P3 notes that this visualization naturally represents the "hierarchy" of their work, and could be taken further by supporting nested forks. P6 reflects on how this would save them from spreading their "explorations across multiple Jupyter notebooks" which must then be carefully "organized in a certain way."

7.6 Threats to Validity

Our study has three threats to external validity. First, the participants did not do tasks that were important to them, on their own data, in their own time constraints. We selected datasets that would be sufficiently complex for participants, giving all but 2 of them enough time to reach some meaningful decision points. Second, the study's short duration limited the complexity of analysis participants were able to perform. That being said, people felt engaged in the task and were still able to explore the data and the tool in order to give meaningful feedback. Finally, participants were all members of a single organization.

8 DISCUSSION

This paper introduces a tool that expands upon the linearity of notebooks. Visually, it breaks the single-column view of cells. It also exposes multiple, independent execution states and allows users to move back through them instead of simply forward. We observed that users find utility and desire to incorporate forking and backtracking within their data analysis workflows. The positive reactions from this initial implementation, along with insights into how users chose to leverage this functionality, suggest that further exploration will be fruitful. Future work can investigate more effective interfaces to manage branches, compare various paths, and manage the tension between linear history and non-linear exploration.

We found that these ideas map well to users' expression of alternatives and decision points. In the usability study, participants used forks to express and compare different approaches, from finding which visualizations make the best sense of the data to comparing the effectiveness of different machine learning models. Though side-by-side comparison had some challenges, the ability to proximally compare was crucial. We also found users interpreting the results from one exploration to go back to a previous cell and branch off to explore another approach, although they often did not use backtracking to express that intent.

We found participants adopted this tool to address other needs as well, in particular, as a debugging tool. Forking allows users to create a "scratch pad" to make sense of their data or to learn library functionality. For two participants, backtracking allowed them to undo otherwise difficult-to-reverse state changes, such as corrupting the data.

Traditional breakpoint debuggers and the RStudio [8] environment provide windows to show the current values of variables. In contrast, the existing notebook interface keeps this execution state hidden, unless users write scripting code to query the state. In hindsight, adding a variables window may have cleared up our participants' confusion about the active state on each path of a fork. In addition, a variables window would have provided an affordance for users to browse to past execution states, which would have made the backtracking feature more prominent.

Finally, this work highlights some challenges that must be addressed before this tool could be truly effective for the length and complexity of real-world data science tasks. If multiple forks are allowed, more work must be done to explore how to best visualize and help users manage the full hierarchical structure. For example, a simple interface to hide paths, such as tabs as in a browser, as well as small hierarchical viewer of forking points could address much of this challenge. Alternatively, a more sophisticated interface might be effective, such as one inspired by Variolite [11]. Better visualization and management could also address the confusing semantics to "above-fork" out-of-order cell execution, for example by simply forking automatically for users, though more work would need to explore if this is sufficient.

9 CONCLUSION

Based on an analysis and interviews with 6 professional data scientists, we have developed a prototype that explores forking and backtracking. We evaluated this prototype with 11 professional data scientists, confirming that forking, as a metaphor, is a helpful way to annotate decisions and compare alternatives. Backtracking, or an ability to navigate back through previous executions, and forking also address debugging challenges which arise from the highly iterative nature of notebooks. Data scientists responded positively to the forking and backtracking tools, which provide them access to a more powerful notebook experience. Their interactions highlight UI design opportunities around managing forks and paths to create a more scalable experience.

[8] https://rstudio.com/

ACKNOWLEDGMENTS

We thank Gonzalo Ramos and Souti Chattopadhyay for the fruitful discussions, and the data scientists at Microsoft for participating in the studies.

REFERENCES

[1] Sara Alspaugh, Nava Zokaei, Andrea Liu, Cindy Jin, and Marti A Hearst. 2018. Futzing and moseying: Interviews with professional data analysts on exploration practices. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 22–31.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, Hoboken, NJ, USA, 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[3] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706

[4] Erin Cherry and Celine Latulipe. 2014. Quantifying the Creativity Support of Digital Tools through the Creativity Support Index. *ACM Transactions on Computer-Human Interaction* 21, 4 (Aug. 2014), 1–25. https://doi.org/10.1145/2617588

[5] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. 2012. Debugger Canvas: Industrial experience with the code bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Hoboken, NJ, USA, 1064–1073. https://doi.org/10.1109/icse.2012.6227113

[6] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. 1996. The KDD process for extracting useful knowledge from volumes of data. *Commun. ACM* 39, 11 (Nov. 1996), 27–34. https://doi.org/10.1145/240455.240464

[7] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A. *Proceedings of the VLDB Endowment* 13, 5 (Jan. 2020), 588–601. https://doi.org/10.14778/3377369.3377370

[8] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology* (Monterey, CA, USA) *(UIST '08)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/1449715.1449732

[9] 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 - CHI '19*. ACM Press, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[10] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2917–2926. https://doi.org/10.1109/TVCG.2012.219

[11] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[12] 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 - CHI '19. ACM Press, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300322

[13] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Hoboken, NJ, USA, 25–29. https://doi.org/10.1109/vlhcc.2017.8103446

[14] 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 - CHI '18*. ACM Press, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

[15] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[16] 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 (2019), 66–76. https://doi.org/10.1109/tvcg.2019.2934593

[17] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-Versioning Tool to Support Experimentation in Exploratory Programming. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 6208–6219. https://doi.org/10.1145/3025453.3025597

[18] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How Data Science Workers Work with Data. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, New York, NY, USA, 1–15. https://doi.org/10.1145/3290605.3300356

[19] Nicholas Nelson, A. Sarma, and A. Hoek. 2017. Towards an IDE to Support Programming as Problem-Solving. In *PPIG*. PPIG, Delft, NL, 15.

[20] P. Pirolli and S. Card. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*. ACM Press, New York, NY, USA, 2–4. https://phibetaiota.net/wp-content/uploads/2014/12/Sensemaking-Process-Pirolli-and-Card.pdf

[21] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, Hoboken, NJ, USA, 1–2. https://doi.org/10.1109/jcdl.2017.7991618

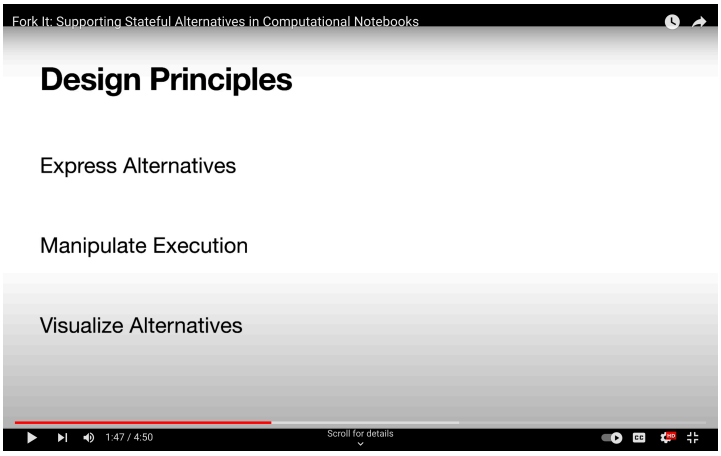
[22] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (Nov. 2018), 1–12. https://doi.org/10.1145/3274419

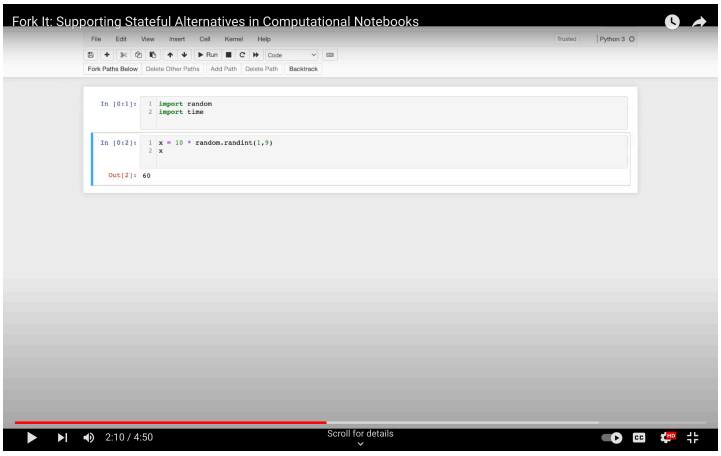
[23] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM Press, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

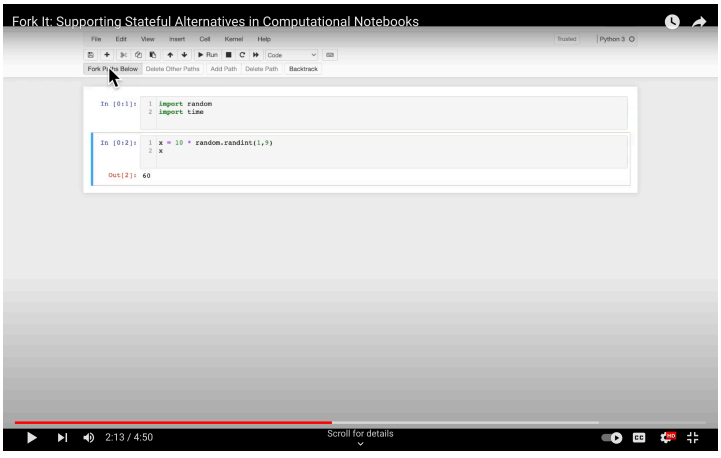
[24] Kluyver Thomas, Ragan-Kelley Benjamin, Perez Fernando, Granger Brian, Bussonnier Matthias, Frederic Jonathan, Kelley Kyle, Hamrick Jessica, Grout Jason, Corlay Sylvain, and et al. 2016. Jupyter Notebooks: a publishing format for reproducible computational workflows. *Stand Alone* 0, Positioning and Power in Academic Publishing: Players, Agents and Agendas (2016), 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

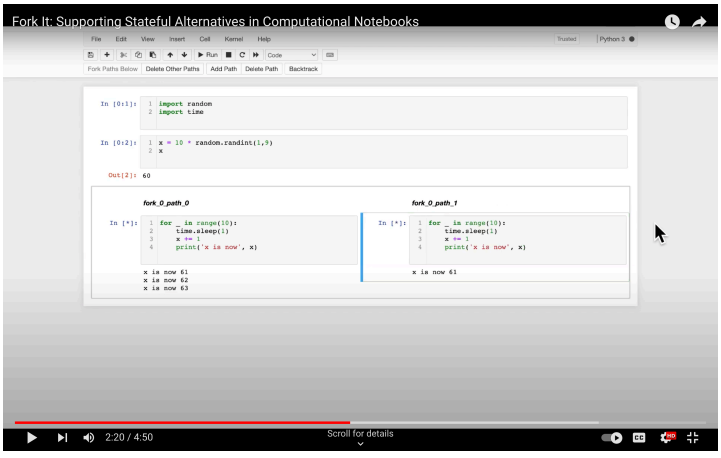
[25] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984), 352–357. https://doi.org/10.1109/tse.1984.5010248

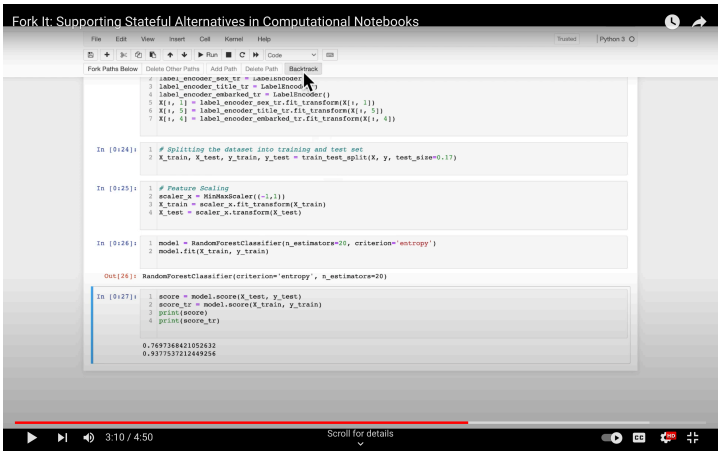
[26] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 2014. Best Practices for Scientific Computing. *PLoS Biology* 12, 1 (Jan. 2014), e1001745. https://doi.org/10.1371/journal.pbio.1001745

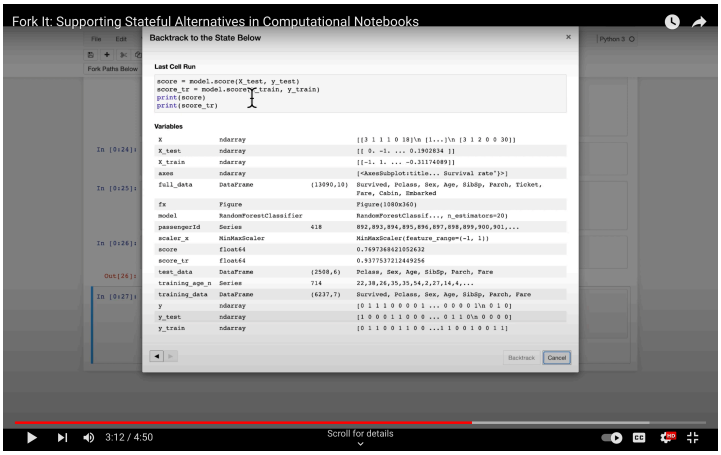


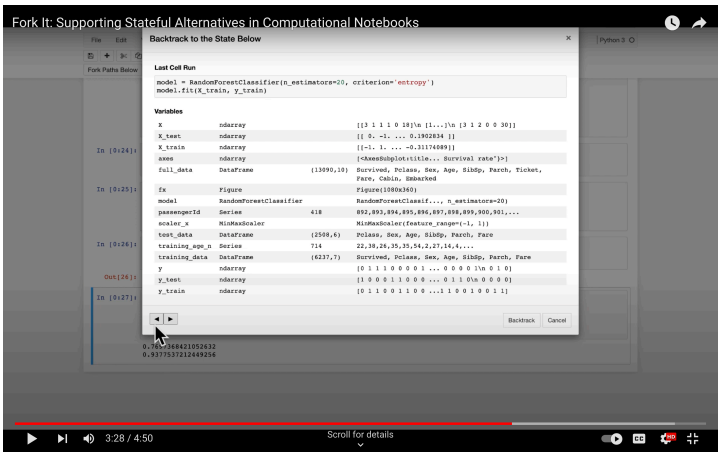


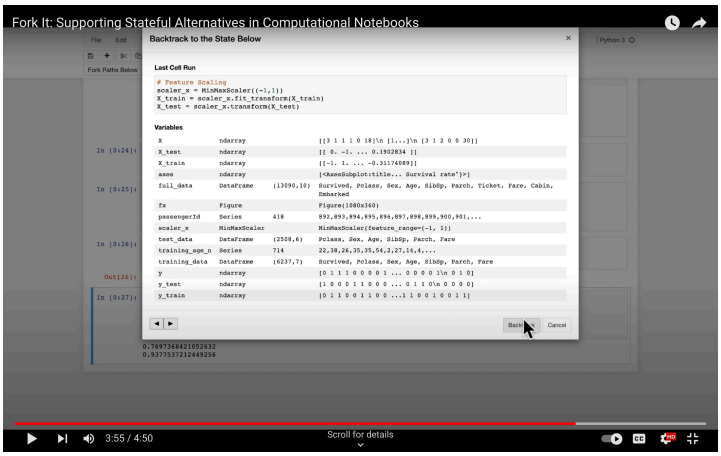


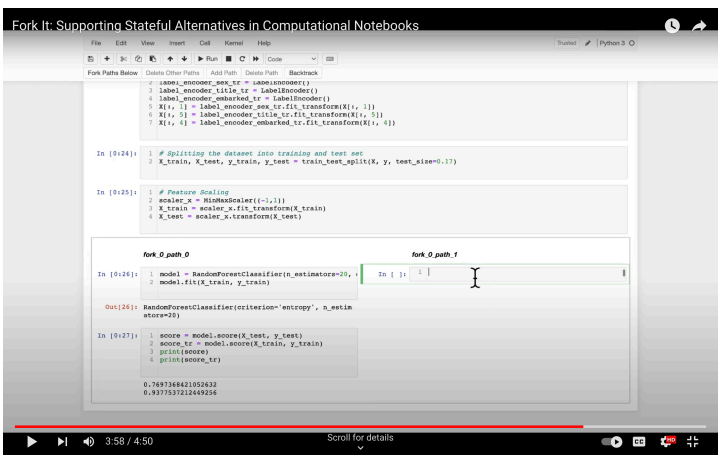



Fork It: Supporting Stateful Alternatives in Computational Notebooks
3:10 / 4:50
Scroll for details


Fork It: Supporting Stateful Alternatives in Computational Notebooks
Backtrack to the State Below
Last Cell Run
Variables
3:12 / 4:50
Scroll for details


Fork It: Supporting Stateful Alternatives in Computational Notebooks
Backtrack to the State Below
Last Cell Run
Variables
3:28 / 4:50
Scroll for details


Fork It: Supporting Stateful Alternatives in Computational Notebooks
Backtrack to the State Below
Last Cell Run
Variables
3:55 / 4:50
Scroll for details


Fork It: Supporting Stateful Alternatives in Computational Notebooks
fork_0_path_0
fork_0_path_1
3:58 / 4:50
Scroll for details