Jupyter's Archive: Searchable Output Histories for Computational Notebooks

Kunal Chaudhary
Andrew Head, Ed.
Björn Hartmann, Ed.

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-72
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-72.html

May 17, 2019

Jupyter's Archive: Searchable Output Histories for Computational Notebooks

by Kunal Chaudhary

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

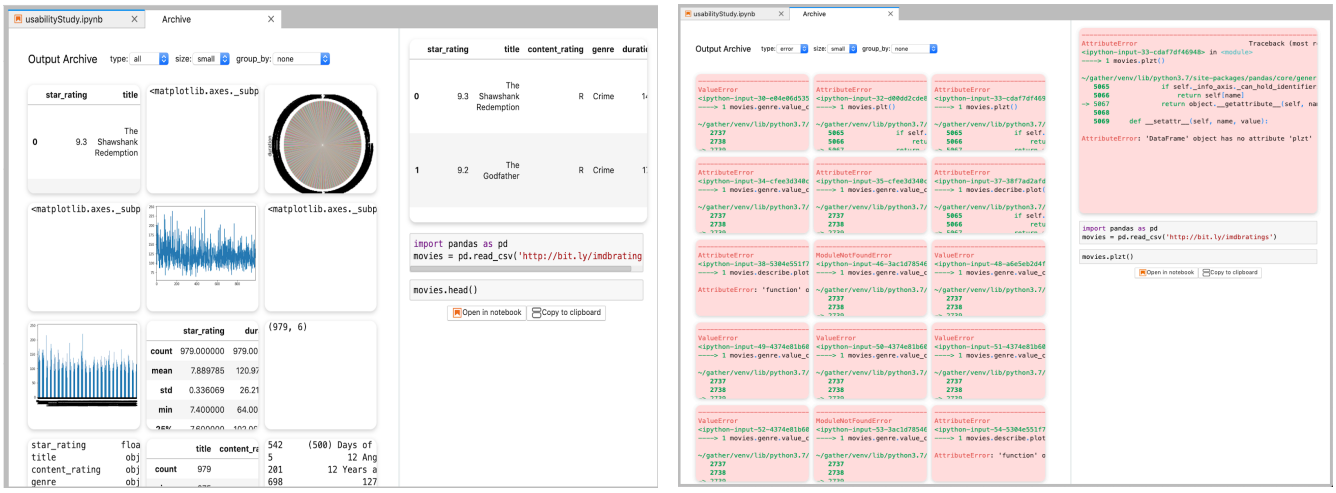
Professor Björn Hartmann
Research Advisor

May 17, 2019
(Date)

* * * * * * *

Professor Armando Fox
Second Reader

5-11-19
(Date)



Jupyter's archive: search through all past outputs generated in a notebook

Jupyter's Archive: Searchable Output Histories for Computational Notebooks

Kunal Chaudhary

ABSTRACT

When using a computational notebook, programmers tend to run, overwrite, and delete cells many times. These actions, which are core to exploratory programming, tend to create a long history of outputs that become fragmented and difficult to track. These outputs are critical to returning to past states when programmers make mistakes in implementation. They are also critical to understanding the evolution of a notebook which can help programmers improve how they code in different situations. To resolve this, this paper introduces the Output Archive, a thumbnail-based output history built into Jupyter Lab that automatically records all outputs produced over the lifetime of a notebook and makes the code that produced them available. This paper also introduces a new class of grouping filters which allows users to navigate large output histories by clustering outputs based on similarities in their underlying code (similar function name, object names, parameters). To test the tool, a usability study was run on 12 computational notebook users who found the Output Archive useful and were able to use its accompanying grouping filters to quickly find important outputs.

INTRODUCTION

When a programmer wants to explore a new data set, implement algorithms, or test different hypotheses, they use computational notebooks. These notebooks allow programmers to run individual pieces of their programs independently of each other. This is a critical affordance in exploratory programming because it saves programmers time and enables programmers to iterate and improve their code at a much quicker pace. For example, if a programmer has just trained multiple neural networks, instead of re-running the entire script to change how one of the models was visualized, in a notebook they could just change the plotting code without affecting the training code. The specific feature that enables this affordance is the cell structure of computational notebooks. Each cell is a runnable script that enables a user to break up their program into modular chunks.

Often, each time a programmer runs a cell, they create an output (table, graph, text, etc.) in order to check that their code is working as intended or in order to visualize some part of the data set for analysis. As the programmer tweaks and improves their code in a notebook, they quickly produce a "large number" of outputs that becomes "laborious" to sort through [1]. These past outputs produced are vital to tracking "which steps" in the notebook "lead to

which results", or to recover previous states in the notebook [2].

Unfortunately, there has been a lack of useful tools that help computational notebook users keep track of the outputs they generate. Generally, notebook users resort to using "version control tools", copying "scripts" and "outputs", or commenting out code in order to take snapshots of the notebook's state [1]. Recent work in this field, however, has proved more promising with the advent of output history extensions that enable users to explore different output variants [1]. These recent tools, however, only offer histories relevant to specific cells and fail to look at the relationships between outputs spread through the entire notebook.

In this paper, we aim to improve how notebook output history is preserved, presented, and searched. We introduce a tool that automatically organizes and makes past outputs searchable. We use a thumbnail-based interface to display all outputs ever produced in a notebook. This tool is built on top of *code gathering tools* which associates outputs with the exact slice of code that produced it [9]. A slice is a mini-program that only contains the code that led to a specific output. Without slices, each output would be associated with the entire notebook that produced it, making it tedious or even impossible for the programmer to recover the relevant code.

Other techniques leveraged in implementation include Abstract Syntax Tree (AST) node traversal, program parsing, and AST diffing. These techniques allow us to not only compare the lines of code that produced an output, but also break apart those individual lines to find similarities.

This paper makes two unique contributions. The first is the design and implementation of the *Output Archive* for Jupyter Lab, an interface for Jupyter Notebook which is an open-source notebook used by millions of people [3]. This tool allows computational notebook users to quickly recover the exact slice of code that produced any past output. A key feature that enables quick output recovery is our new class of grouping filters that enables a user to group outputs by similarities in the code that produced them. For example, with our group by function name filter, a user can find the name of the function that produced a unique plot even after they overwrote the code preceding the plot. The unique insight powering these grouping filters is that users can see valid relationships between outputs produced across different cells, instead of just in the same cell which is what current tools focus on [9]. This more flexible definition of variants, or the different versions and types of outputs, and similarity in outputs allowed us to achieve a large dimensionality reduction that made output search easier

Users looking to recover a past output merely have to click on a toolbar command which visualizes the outputs (text, tables, graphs, errors, etc.) generated and search through the outputs using filters.

The second contribution of this paper is a controlled usability study that explores the usability and usefulness of the Output Archive and its filters in navigating large output histories. 12 notebook users from a variety of backgrounds ranging from students to software engineers participated in an in-lab study to recover past outputs produced by a notebook coding session. We discovered that not only was the Archive useful, but the filters we developed enabled easier search. On top of this, we discovered a variety of ways programmers would like to group and search their outputs, which revolved around visualizing different types of variants. For example, one variant that participants stated they would like to see is grouping outputs that were produced by the same data set.

RELATED WORK

Software history, the history of code executions (logs of executions), has been a core focus of academic research for many years. Software history can help exploratory programmers recover from mistakes and learn more about their code. Unsurprisingly, over 80% of programmers find software history useful while developing software [4].

In order build a history for computational notebooks, we studied past work in making histories for other coding systems. Some approaches have instrumented entire operating systems in order to record and visualize all past activities relevant to code generation (e.g. creating code, visiting a coding website, viewing an image) [5]. Other approaches have narrowed their scope by just instrumenting an underlying programming language like Python and recording all function calls and associated parameters [6]. Most relevant to our work are tools that instrument the code editor to provide useful features like the ability to undo changes back to a previous state of code [7]. All of these approaches have some way of grafting tools for logging and displaying history onto existing coding environments and making the history accessible to programmers. We took inspiration from these approaches and decided to introduce an extension to computational notebooks that managed the logging and displaying of history.

In order to build our notebook history, we had to figure out what to record. One approach in this space revolved around preserving an individual cell's code history and enabling programmers to swap a cell with previous versions of itself [2]. Another tool took this same approach except dove deeper in the cell and recorded and presented histories of individual, user versioned scripts [8]. The tool which we built on top of, *code gathering tools*, also enabled users to see all previous versions of a cell's code when examining code slices [9]. This past work informed the overall design our Output Archive architecture, but we deviated from their code-focused histories when creating our output-focused history because our primary goal was to help programmers find past important outputs.

The tool our output-focused history most closely resembles is Verdant, which is a plugin that records history of all outputs produced by a specific cell in a notebook. This tool provides a graphical interface for navigating a specific cell's past outputs [1]. While we share the same type of output history, we chose to present our history in a way that was more optimized for search.

The primary purpose of our output history is to enable users to find a past output and its associated code effectively. This meant that we had to take a large history and provide different ways to filter it in order to make searching easier. Clustering outputs only by their common cells restricted the different types of filters that could be created. Past work has suggested that there exist types of variants that users care about besides outputs or code versions that originate from the same cell [9]. We recognized the validity in this and structured our Output Archive so that our default history view showed all outputs generated in the notebook in unfiltered rows and columns.

With our unstructured, default history interface, we set out to explore how we could cluster the outputs in ways that make sense to the user. We took inspiration from other tools that cluster code, such as those in the education realm, where past attempts have successfully clustered different types of homework code into a navigable user interface [10]. In the tutorial realm, approaches have used clustering to make it easier to navigate a vast body of tutorials [12] [13]. In software engineering, tools have focused on clustering GitHub code in order to find usable code snippets using abstract syntax tree (AST) traversal [11]. The existence of these different approaches in different domains motivated us to use clustering algorithms in our grouping filters and to implement them with AST traversals. Furthermore, these approaches demonstrated that the best way to cluster bodies of work/code was to focus on clustering specific subsets of code.

The final area we explored was how users traditionally found the code that produced an output. Previous attempts have been made at tracing a program output back to the code that caused it [14], or using visual effects on web pages to find the web code that produce it [15]. These attempts showed us that finding the most granular code that caused an output was important, so we focused on associating notebook outputs with the most relevant lines of code possible, their slices.

EXPLORATORY ANALYSIS OF OUTPUTS AND VARIATION IN COMPUTATIONAL NOTEBOOKS

Before we implemented our Output Archive, we analyzed the outputs of 25 notebooks from a random sample of 1000 notebooks of the UCSD Jupyter Notebook GitHub Dataset which contains approximately 1.25 million Jupyter Notebooks [16]. We felt this was necessary because we wanted to base our implementation and design decisions in data, instead of anecdotal evidence. This data set was a snapshot of all available Jupyter Notebooks on GitHub as of July 2017. We analyzed these notebooks in order to answer two primary questions that would help us craft grouping algorithms that would make it easier find important outputs:

Q1) *How is the code that produces outputs distributed throughout cells?* Are there usually multiple outputs per cell? Or only 1?

Q2) *What are the different ways a programmer might want to group outputs in an output archive?* Do we have to look at the outputs themselves, or are there similarities in the code that produced them?

How many outputs does each cell have?

The first characteristic we noticed was that cells either contained multiple outputs, or a single output that was typically the last line of the cell. We rarely noticed single output cells where the output was not the last line of the cell. In the following images, paradigm 1 produces one table with the head() function.

```
df['Gendercolor'] = df['Gender'].map({'Male': 'blue', 'Female': 'red'})
df.head()
```

Paradigm 1: Single output in a cell – code that produced that output is last line of preceding cell

In paradigm 2, the 2 plot() functions produce separate plots.

```
males = df[df['Gender'] == 'Male']
females = df.query('Gender == "Female"')
fig, ax = plt.subplots()

males.plot(kind='scatter', x='Height', y='Weight',
           ax=ax, color='blue', alpha=0.3,
           title='Male & Female Populations')

females.plot(kind='scatter', x='Height', y='Weight',
             ax=ax, color='red', alpha=0.3)
```

Paradigm 2: Multiple Outputs in a cell.

From an analysis of the 25 random notebooks, we saw that paradigm 1 accounted for roughly 80% of all outputs produced. This meant we could focus on only grouping

paradigm 1 outputs (code that produced output is directly above the output and is the last line of the preceding cell) and still produce useful groupings. This was fortunate because we were unable to properly slice multiple outputs in the same cell because our slicing algorithm provided by *code gathering tools* wasn't set up to handle that case.

How can we group outputs together?

The next thing we looked for in these notebooks were different ways to group outputs together in order to capture different forms of variants. These different forms of variants would allow us to filter huge lists of outputs into groupings that made finding important outputs easier.

From the outset, we decided to capture variants by grouping outputs that were similar in some way. This presented a design challenge because it's not immediately obvious what defines similarity. Outputs could be similar in their content, data type, or underlying data set.

In looking at the code of these outputs, we noticed that they differed primarily in 3 ways: their object, function, or parameters (Figure 10):

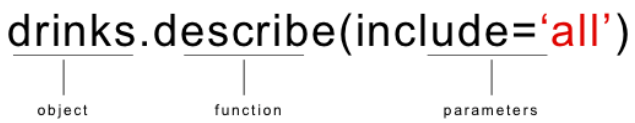


Figure 10: The three parts of code that causes outputs

Taking this observation, we found many examples in the data set:

```
Image(filename='img/37.png')
```

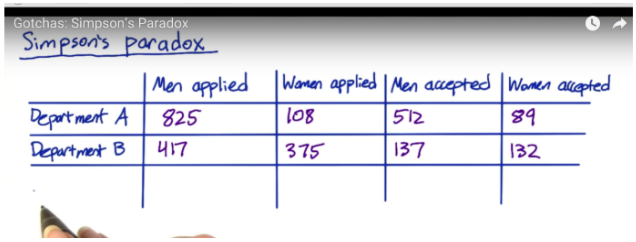


Figure 11: 37th Image printed out in a notebook

```
Image(filename='img/38.png')
```

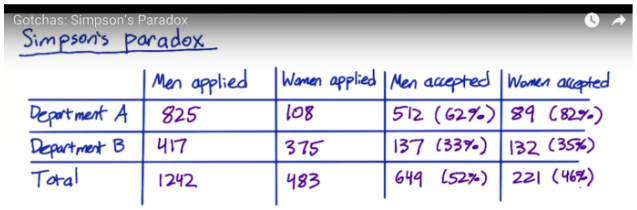


Figure 12: 38th Image printed out in a notebook

In these two outputs, the programmer modified the parameters of their Image() function to produce two images of a table that shared similar values (Figure 11 & 12). We saw that examples like this, where the programmer only modified the parameters, object, or function, occurred at a high frequency, so we chose to capture similarity through a simple heuristic: *Similar outputs might share similar code.*

While this heuristic made groupings that were understandable to a programmer, this definition also produced some groupings that were less clear:

```
Image(filename='img/40.png')
```

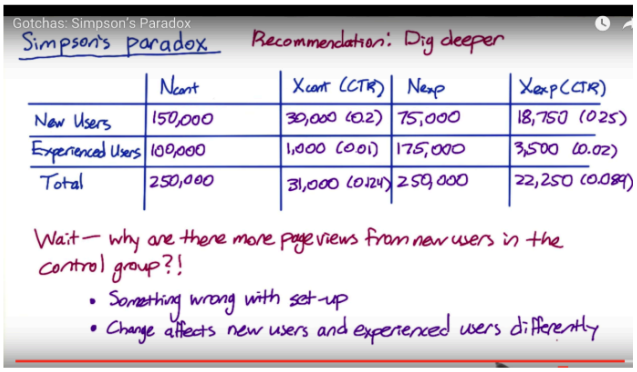


Figure 13: 40th Image printed out in a notebook

```
Image(filename='img/41.png')
```

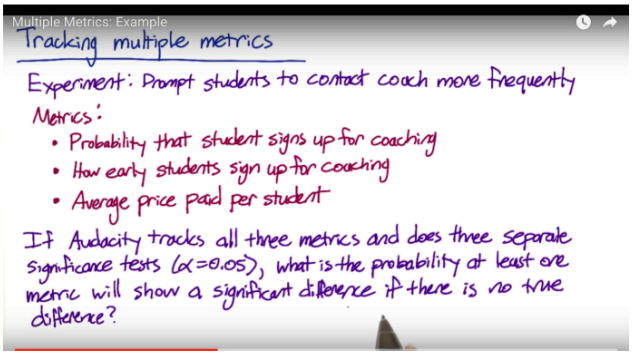


Figure 14: 41st Image printed out in a notebook

Although these two images have different contents, they are similar in data type (both are images), and they might share a similar purpose: displaying all the images of an image library (Figure 13 & 14). Ultimately, even if our heuristic creates groupings that are unclear or false positives, we were satisfied because it increased our chances of capturing axes of similarities that we couldn't predict beforehand.

We structured our grouping algorithms to capture how the code that causes outputs differed in objects names, function names, and parameter names.

To get a sense for how often we could properly group outputs using the similar code heuristic, we took all the pairs of outputs generated by our heuristic and divided it by the total number of possible pairings (total number of permutations of outputs) in each of the 25 notebooks. We then averaged this percentage across all notebooks. In the end we found that our grouping heuristic had the capacity to group together 41% of all possible pairings on average in a notebook. This number is relatively high because many of the notebook in the data set were relatively short and created for very specific purposes (e.g. printing 5 plots that were related to each other).

While many of the other possible groupings may not make sense, this percentage shows that there exist other approaches that could capture different types of variants. One such type of variants could be outputs that had similar input data, but different output code. Another type of variants not captured by our heuristic could be percentage of lines in the respective code slice that were the same. These cases are common enough that they could each easily cover over 5% of all possible pairings.

We ultimately chose to implement only a subset of the grouping algorithms we came up because they didn't appear with enough frequency. Another reason is that we didn't want to overwhelm the user with too many grouping filters.

DESIGN MOTIVATIONS

The initial motivation to build an Output Archive stemmed primarily from a study of past work done in this space [1][2][9]. In each of these studies, users consistently expressed desire for some sort of version control for computational notebooks. Those users also expressed an interest in having features that allow them to recall how something was produced in a way that current versioning tools don't catch. We outline in this section the specific motivations behind various design decisions.

Group outputs based on the data transformations that produced them, not the cell they come from. We wanted some way to achieve dimensionality reduction from the unfiltered view in order to make search easier. Past work in this space has explored grouping outputs by the cell they were generated in, but we analyzed a 25 notebooks at random in the UCSD Jupyter notebook dataset and found that outputs generated in different cells shared many similarities and variants that would make sense to a user if clustered together [16]. Ultimately, we felt that grouping based on underlying code and not location in a notebook would achieve groupings that could capture different types of variants, especially considering cells can get overwritten.

Guide exploration with output thumbnails, not code. Faced with the reality that notebooks users generate a plethora of outputs during their coding sessions, we required a condensed way to display all the outputs in an unfiltered, top-level view. One alternative we considered was just listing the code that caused each output in a table view. From our body of research, we discovered that programmers generally look at outputs, changelogs, and source code when they are navigating an overabundance of code [18]. This inspired us to try out the visual output approach as an alternative to the source code and changelog approach.

INTERACTING WITH THE OUTPUT ARCHIVE

In order to understand the user experience of the Output Archive, we will walk through a short scenario using Jupyter Lab. Our scenario is centered around a programmer named Abe who is working with an IMDB movie ratings data set. His goal is to understand different characteristics about popular and unpopular movies, like their genre, duration, and ratings. Over the course of a few hours Abe generates hundreds of different outputs from dozens of different cells. These outputs include text, errors, tables, and images.

At some point in the present, Abe tries to recall an output that he previously produced which counted the number of PG-13 movies in his dataset. He's long since overwritten the initial cell that produced this output and he only vaguely remembers pieces of the code he used to generate that output. He anticipates the result will be hard to reproduce. Eager to retrace his steps, he turns to the Output Archive for help.

Opening up the Archive and an initial scan of outputs

Abe clicks on View Archive in the plugin toolbar on Jupyter Lab (Figure 1). A separate notebook tab, labeled "archive", appears and he is met with an unfiltered archive that contains all outputs ever produced in his notebook, even if the kernel was restarted at any point (Figure 2).

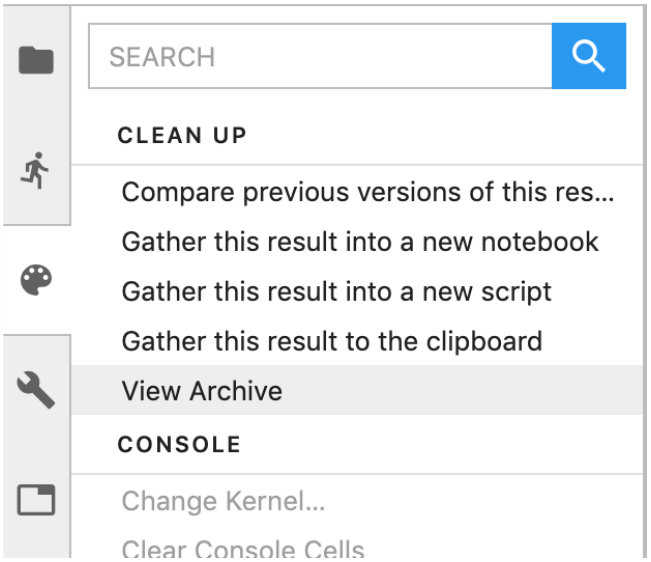


Figure 1. Plugin toolbar on Jupyter Lab

He scrolls through to see if any of the output thumbnails are immediately obvious to his task at hand. He notices that these output thumbnails are scaled down so that many of them fit on his large desktop monitor.

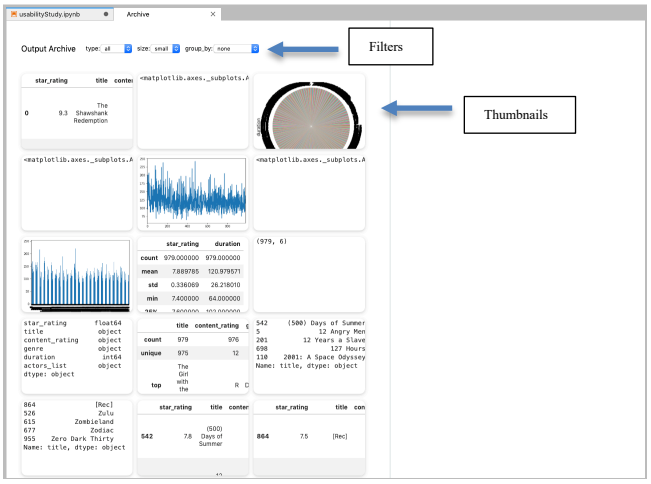


Figure 2. Initial archive view

He also notices that the outputs are listed in reverse chronological order so that he can access the most recently generated cells more easily than the older cells. He clicks on a few of the outputs to get greater detail about the exact lines of code that caused the selected output (Figure 3). Upon clicking on an output, a full-size scrollable version of the output and the ordered subset of code that produce it appears.

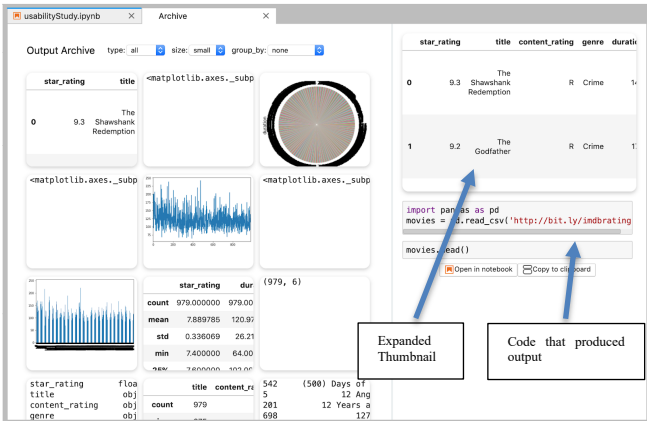


Figure 3. Clicking on a thumbnail

He notices that he could export this code to a new notebook if he wanted to recreate the output. After doing this a few times with different outputs, he remembers that he's looking for some sort of text output.

Trying the first few type filters

Abe notices the type filters at the top of the Output Archive. He clicks on the type filter and sees that he can select table, error, text, and image (Figure 4).

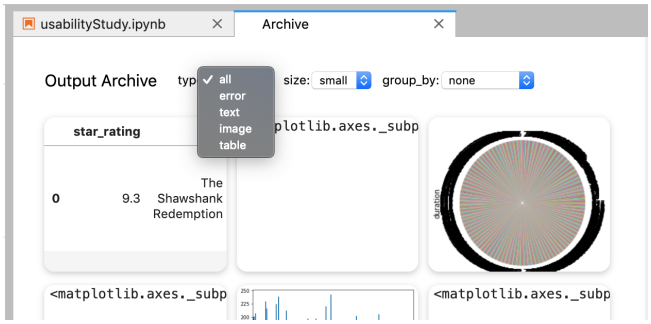


Figure 4. Clicking on type filters

He clicks on error because he wants to see how many he's produced over his coding sessions and his Archive narrows down to 15 errors (Figure 5). Abe realizes that this view could be useful if he ever runs into an error and wants to remember how he dealt with it in the past.

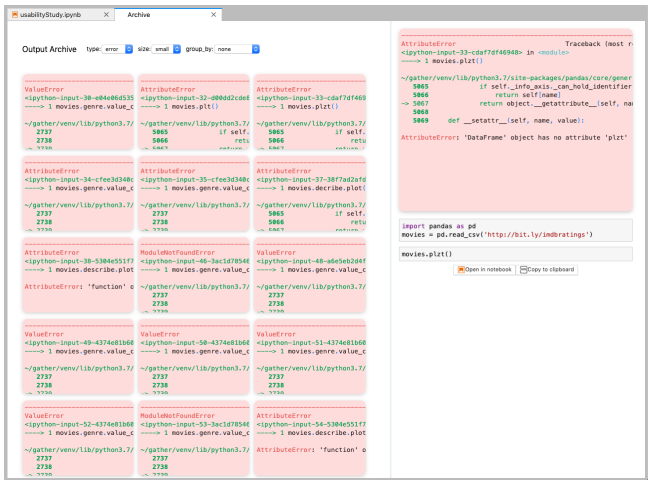


Figure 5. Clicking on type filter: error

Abe, remembering that he's looking for a text output, uses the type filter to narrow down his selection to just thumbnails that show text outputs (Figure 6). Unfortunately, he realizes that he's still dealing with over 70 different outputs.

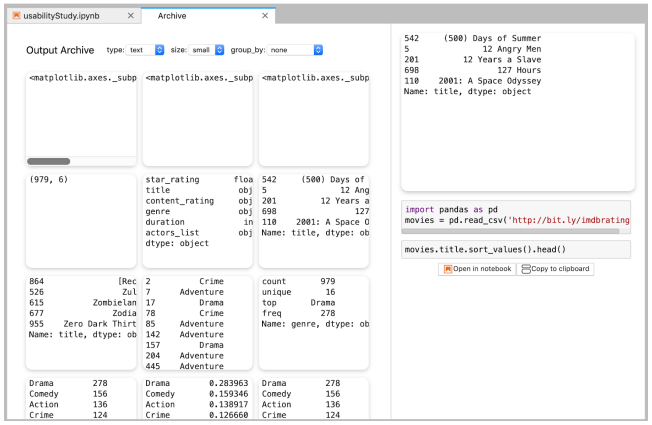


Figure 6. Clicking on type filter: text

70 Outputs is ultimately too many to parse through individually, especially since they were all produced for different purposes. After exploring some more, he gets a feel for how some of his older output code was structured. He then realizes that he used some sort of count function to produce his text output.

Using grouping filters to group by code

Abe decides to use the group by function name filter in conjunction with the text filter (Figure 7).

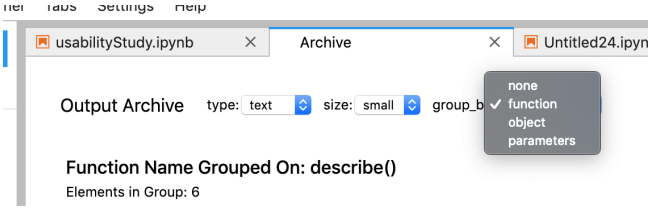


Figure 7. Clicking on group_by filter

Abe scrolls down to find a function called value_counts() (Figure 8).

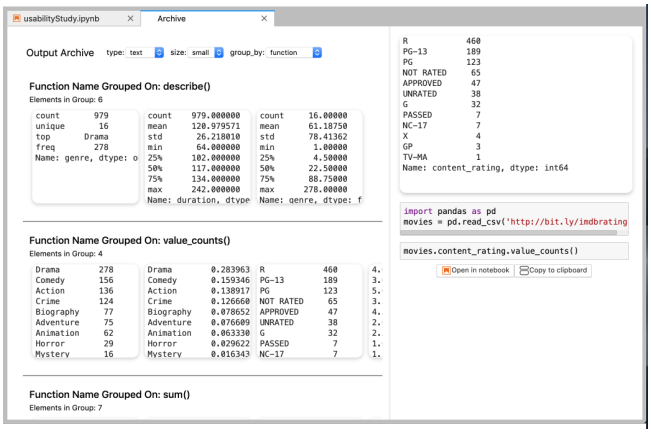


Figure 8. Clicking on group_by filter: function

After scrolling horizontally for a moment, he sees a tally of the number of films that have specific ratings. He's arrived at his output and decides to replicate this exact slice of code so he can do further manipulations on the data set. He exports it to a new notebook (Figure 9).

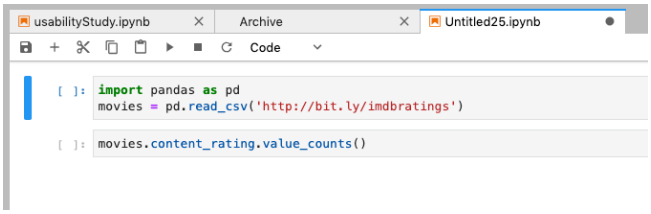


Figure 9. Export code to new notebook

IMPLEMENTATION

The Output Archive was built on top of *code gathering tools*, an existing plugin for Jupyter Lab [9]. The *code gathering tools* provided a convenient way to associate each output with its associated slice of code. The Output Archive was implemented with roughly 1,000 lines of TypeScript code. This code covered both tweaks to the program analysis code and the user interface implementation.

Persistent History

The *code gathering tools* extension provided an execution log of all cells. These cells contained various pieces of information like id, text, and outputs. Each time a cell is run in a notebook, it is stored in *code gathering tool's* execution log.

One of the initial challenges we faced was creating a persistent history that exists after a user leaves Jupyter Lab. Wanting to avoid introducing an external storage repository, we discovered that Jupyter Lab maintains a metadata associated with each notebook that persists between individual sessions. This implementation decision makes notebooks more shareable because the history is preserved in the metadata, and it allows someone to open up a notebook years after its creation and still have access to all the outputs produced and the code that produced them. To take advantage of this feature, we implemented a serialization of a notebook's entire execution log (execution time, code, output) on save, and deserialization of it on open. Another obstacle we faced in making a persistent history was the fact that Jupyter notebook changes the ID of cells on different kernel sessions. To solve this, we had to create a persistent ID stored in the metadata of a cell that would stay constant even if the default cell ID changed.

Visualization

The front end is implemented as a Widget, a building block UI element of Jupyter Lab. The frontend was developed with large laptops and desktop displays in mind and utilizes different jQuery events to handle the filtering and clicking features. The filters function primarily based on a class tagging system. As each widget is generated, it's pre-tagged with a type of output (table, image, text, etc), and a specific type of grouping. As a user clicks on different filters, the UI hides all elements that don't have the specified class. This

provides a large performance gain while using the Output Archive because the grouping filters don't need to run every time a user clicks on them. There is a setup cost of 1-5s in order to properly load and sort all of the different outputs. This cost occurs only when archive is initially loaded. (this could easily be folded into the initialization of the notebook itself and can be unnoticeable to the user.

Grouping Filters

In order to develop our grouping algorithm, we analyzed 25 random notebooks from the UCSD Jupyter notebook dataset and discovered two primary output paradigms (refer to Paradigm 1 and Paradigm 2 images in Exploratory Analysis section) [16].

We ultimately chose to target outputs that fell into Paradigm 1 (code that produced an output was in the last line of the preceding cell, and the output was the only output produced by the preceding cell). In order to compare the code that produced outputs with one another, we had to compare their ASTs.

We developed three grouping algorithms for objects, function names, and parameters that traversed their AST and then fed an inner node into a tree diffing algorithm called Zhang-Shasha [18]. More specifically, each grouping algorithm performs an equality check on only one part of the code. In order to effectively work with Zhang-Shasha, we had to convert our ASTs into an appropriate format. This required us to modify *code gathering tool's* underlying parser to insert parseable labels into each node. This approach allowed us to generate clusters based on equal object names/chains of object transformations, equal function names, and equal parameter names (Figure 15-17).

```
movies.content_rating.value_counts()
```

```
movies.genre.value_counts(normalize=True)
```

Figure 15: Cells grouped together by function name value_counts()

```
movies.describe().plot()
```

```
movies.describe().head()
```

Figure 16: Cells grouped together by object name, movies.describe()

```
movies.genre.value_counts().plot(kind='box')
```

```
movies.duration.plot(kind='box')
```

Figure 17: Cells grouped together by parameter name, kind="box"

IN-LAB USABILITY STUDY

We designed a 30 minute, in-lab usability study designed to understand how users interact with the Output Archive. We ran one pilot study with two users to determine the basic comprehensibility and usability of the tool. From these users we discovered that they understood the premise of the tool relatively quickly, so we focused our in-lab usability study primarily on search and utility of the archive. Our

study was designed to answer the following research questions:

RQ1. *Does filtering by type and grouping code in different ways help programmers find important outputs amidst an abundance of past outputs?* In what situations does grouping by variants help? What other types of variants would be useful to group by?

RQ2. *Do users find the Output Archive useful?* How would an Output Archive have helped them in their past coding experiences? Would they use an archive in the future?

We selected 12 participants with coding and computational notebook experience from individuals that the researchers previously knew. This was a convenience sample of participants; however, we took precautions to mitigate bias present from selecting the convenience sample by recruiting participants from a variety of places (industry, research, undergrad) and informing the participants that the facilitator was evaluating the system, but had not created it. Five participants were professional programmers, and seven were students (2 female, 10 male). The median experience in programming on a Likert Scale was 3-5 years and the average age was 21.75 years old. The majority of our participants (8/12) used computational notebook at least on a monthly basis (4 Daily, 2 Weekly, 2 Monthly, 4 Yearly). All participants were offered a $20 Amazon gift card as compensation for the time.

Tasks

Participants began the study by signing a consent form. The study centered around identifying different outputs in a pre-made notebook. This notebook was 50 cells long and had generated 150 different outputs. Participants were told that this was a notebook they had generated over a long coding session that primarily dealt with visualizing and analyzing an IMDB movie ratings data-set. They were then given 4 tasks, each with an explanation of why they needed to find a specific output. These tasks were inspired by the tasks proposed in the Verdant's usability study, which also focused on evaluating computational notebook histories [1]. The 4 tasks, with background information shortened, were the following questions:

1. What are the different ways you visualized movie durations?
2. You've run into an error while trying to make a Kernel Density (KDE) plot to visualize movie ratings. You previously ran into this error when trying to visualize movie genres. Figure out why that previous error occurred.
3. You remember you printed out all the movies that have a runtime of over 3 hours and 20 minutes. Name a movie with that runtime.
4. You're coding and you print out an empty graph. You remember you previously printed out some empty graphs. Figure out what parameters caused those empty graphs previously.

Each participant was timed, and the number of clicks on each feature (type filter, widget, size filter, grouping filter) in the archive was recorded. In order to get a control for the efficacy of our filters, we only enabled filters for two of the 4 tasks (either task 1 and 2 or task 3 and 4). 6 participants were given the study where tasks 1 and 2 had no filters and the other six were given task 3 and 4 with no filters.

Before all 4 tasks, participants followed a small tutorial on how the archive worked. This primarily entailed clicking on different outputs with only context being to familiarize themselves with the available affordances. Before the filtered tasks, participants were given a tutorial on how the different filters worked. After each task, the user filled out a survey that asked three questions about the difficulty of the task and the usefulness of the different archive features in accomplishing the task.

After all four tasks were complete, the users were asked a series of 4 open-ended questions. 3 questions focused on the comprehensibility of the grouping filters. The final question was an open-ended question about the usefulness of the Output Archive. After these questions, the users filled out a demographic survey.

Each participant was given the same 13-inch MacBook Pro with the archive preloaded onto it. This MacBook was connected to a 1920x1080 monitor in which they had the archive opened up in one tab, and the survey opened in another. The study was performed with the Output Archive open on the desktop monitor. The MacBook screen showed the console logs associated with the browser.

During and directly after each task, 4 primary measures were taken to gauge the efficacy and usefulness of various features. These measurements were the time it took to complete the task, perceived difficulty of the task, number features clicked in order to accomplish task, and perceived usefulness of features. For all 4 tasks, we changed whether or not the participant had access to filters. This involved counterbalancing the order of the interfaces while the question order remained the same. This was done to reduce the effect of which interface affected the participant's preferences and task performance. The intended goal of this was to learn more about the efficacy of filters in the archive to aid search.

RESULTS

In this section, we refer to the 12 participants with the pseudonyms P1-P12.

Are the filters helpful in finding important outputs?

Timing

Filters allowed participants to accomplish their tasks at faster speeds to the non-filtered tasks, though the result is not statistically significant (P = .08544, U = 204, Using

Mann Whitney U Test). Although the distributions of the timing data were similar, the timing data did informally trend towards suggesting that filters provided some speed up in certain tasks.

In tasks where the desired outputs were spread throughout the archive (Tasks 1 and 4), the filters appeared to provide on average a 62.5% speed up in search time. In tasks where the desired outputs were clustered together, towards to the top of the archive, or were visually easy to spot, the presence of filters provided a 49% slowdown (Tasks 2 and 3). Although these measurements aren't definitive in showing the efficacy of filters, they do provide a sense of which way the data leaned. Another interesting note is that 8/12 participants accomplished the filtered tasks faster on average than the no-filter tasks.

These informal differences could be attributed to a few reasons. In the absence of filters, users quickly realized that they had to perform an exhaustive search of all relevant thumbnails in the archive in order to find the answer. In Tasks 2 and 3, those outputs were towards the upper half of the archive. In Tasks 1 and 4, the outputs desired were at the top of the archive and at the bottom of the archive, so users had to perform an exhaustive search.

Ultimately although the data suggests filters provided a speed up, it's not a pronounced enough effect to be statistically conclusive.

Perceived Difficulty

Although users had differing performance times based on the existence of filters and the order of activities, they always found tasks more difficult or as difficult when filters weren't available. To measure this, we took their responses to "How difficult was the given task" that they filled out after each activity and assigned each rating from (Hard to Easy) a score from 1 to 5. We then added those scores up and created a perceived easiness rating. Tasks 1 and 4 were perceived to be much easier when filters were around, and tasks 2 and 3 were perceived to be only slightly easier when filters were around. This suggests that the presence of filters caused people to rate tasks as easier even when they didn't use them or even when the task was harder for them to accomplish. This conclusion mirrors many of the comments that participants had during the study when they were asked to do tasks without filters. P2 stated once they realized they had to click through the entire Archive to find the result, "Do I really have to?". Multiple other participants expressed verbal frustration mirroring P2's sentiment when searching without filters.

Number of Features Clicked

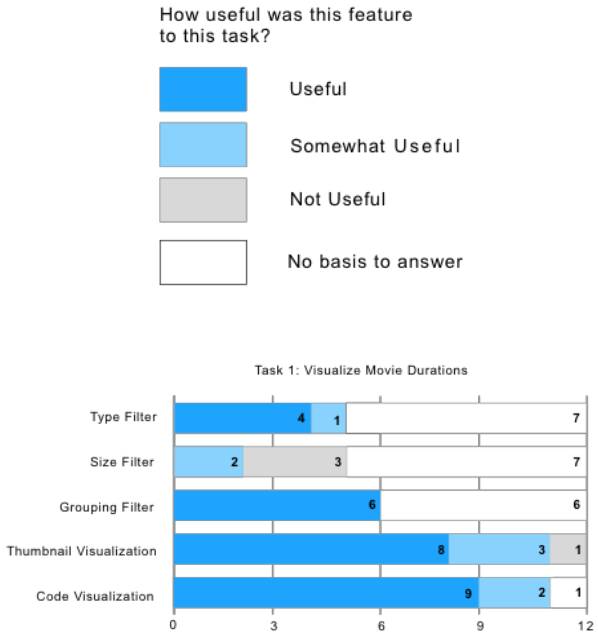
The numbers of features clicked measured the number of thumbnail clicks and filter clicks. The number of features clicked loosely follows the timing measurements. Taken in aggregate, the presence of filters decreased the number of features clicked by 14% in order to arrive to the find the final answer.

Usefulness of Features

When participants didn't have access to filters, they found the thumbnail and code visualization (code snippet that appears when you click on a thumbnail on the right side of the page) most useful for accomplishing a given task. When participants did have access to the filters, the vast majority found the grouping filters useful, with only 3 specific instances of ratings less than somewhat useful. Participants found that filtering by the type of output (table, text, image, error) was useful for all tasks except for the third task. This suggests that the presence of filters for all tasks provided utility. The results of the usefulness measurements are shown listed in Figure 17.

Analysis of Grouping Filters

In general, while participants were able to successful use the grouping filters to find important outputs, they faced challenges understanding exactly what the rules were grouping outputs together. When asked what they thought a grouping filter (object, function, parameter) did, participants offered alternate wordings that they felt better represented the functionality of the grouping filters. For example, many participants thought that the group by



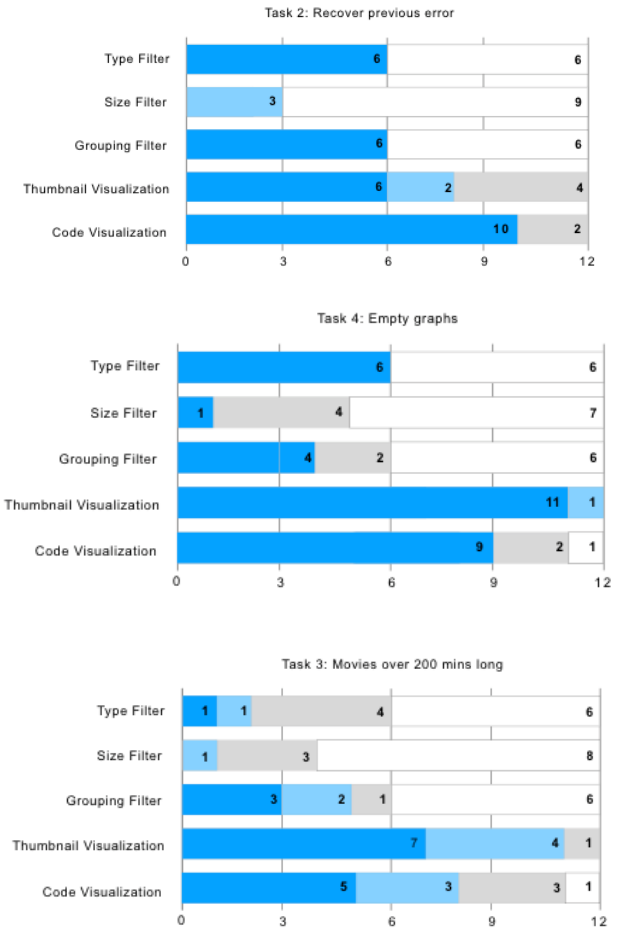


Figure 17: Likert scores for feature usefulness during tasks. Important note: For each task, 6 participants didn't have access to filters.

parameter function grouped on functions and parameters. As P8 puts it, the group by parameter should be more "granular than function", implying that they expected group by parameters to share similarities with group by functions.

Group by function name was the easiest to understand, but difficult to use because many users weren't familiar with the types of functions used in Pandas. One participant stated that they would "be able to better know what the function groupings would exist if they knew how to use pandas" (P3). This led to users having to scroll down with the grouping filter applied in order to understand the full range of function names. Other users coped with this unfamiliarity by using the command-F within page search tool to find keywords in the actual thumbnails. One user stated that they wanted "the function signature to be included" (referring to function definition and explanatory comments) at the top of each grouped section because they weren't familiar with what the functions did (P5).

Group by Object was confusing because of a coding practice in pandas to chain accessor methods and properties in order to create sub objects before the final function call. P5 pointed out that they wouldn't have expected an object to be something that "has a member function called". No participant predicted that the group by object filter would do this. Some participants wanted to see an object tree showing how the parent objects and its sub-objects were related. Furthermore, some users wanted the group by object to not be mutually exclusive in its groupings (e.g. all outputs with the movies object would go in the same group, then it would narrow down as the following groupings were proposed like movies.columns).

Group by Parameters generally seemed to be the least useful filter primarily because users didn't really understand the situations where they would care about grouping on just parameters, without the function names the same. As P2 put it, the group by parameter "seems largely redundant with the group by function". The most important feedback on this filter was that it would be more useful if it grouped by function name and parameters.

Overall although users recognized the utility of the grouping filters, many aspects of the filters need to be better communicated.

Is the Output Archive useful?

11/12 participants stated that this tool would have been useful for them in their past computational notebook usage. 5 participants were so excited about it they asked when the Output Archive as a whole was going to launch. P4 stated that, "this tool is very useful if you don't have great coding practices. A lot of people don't have great coding practices...which is why this is very useful". The coding practices P4 was referring to are keeping an immaculate, append only notebook where all outputs are organized and preserved (no deleting, or overwriting cells).

A few participants explained that they resorted to homebrewed version control methods while using notebooks in the past. These methods included keeping separate notebooks for different approaches or copying and pasting results into an external text editor. P6 stated that they "stopped using computational notebooks in the past because he kept overwriting previous states and got frustrated". Nearly all of the participants mentioned that recovering past states had utility and was something they wanted. Many of the participants used notebooks for classwork in the past and lamented that many times they ruined their notebook by making mistakes and were unable to return to versions of the notebook or cell that worked. This feedback from participants confirmed that Output Archives should be a mainstay feature of computational notebooks.

Design motivations revisited

The usability study validated the choice to make the notebook visually driven based on the fact that participants

found the thumbnail visualization feature useful in nearly every task they were given, with or without filters. The grouping filters provided measurable utility to the end user, even despite the difficulties in education.

Conclusions

The usability study with 12 programmers confirmed that output histories for computational notebooks are useful and that our new grouping filters can help a programmer find a specific output among an overabundance of outputs.

LIMITATIONS

This study and tool had two primary limitations. The first limitation was that our participants didn't write the code that produced the outputs in the usability study. This led to an unrealistic testing environment where users had to figure out the relationship between outputs and code without much context. It's important to note however, this situation could mirror a programmer coming back to their own notebook months after making it. The other primary limitation of this tool lies in the fact that the underlying slicing algorithm didn't slice the immediate cell that produced the output. For example, if a cell had multiple outputs, each output would include that cell in their slice, even if everything in slice, sans the cell, was different. This prevented us from generalizing the algorithm to 20% of output cases.

FUTURE WORK

The following outlines areas for future improvements to the Output Archive.

More Grouping Filters

The current grouping filters only captures a subset of possible variants in a notebook. Other types of grouping filters could capture important qualities about the code associated with outputs. These other types of filters could include filtering by loaded in data source, slice similarity, or cell location.

Notebook-First Code or Output Search

A programmer with intimate knowledge of their notebook and the code inside it may not want to go to the Output Archive to recover an output. For example, if a programmer knew that they were looking for a specific function, instead of going to the Output Archive and clicking the group by function filter, they may just want to click on an instance of that function in their own notebook to bring up the relevant histories. An area for future work is to integrate the Output Archive directly into a working notebook instead of as a separate tab.

REFERENCES

1. Kery, Mary Beth, et al. "Towards Effective Foraging by Data Scientists to Find Past Analysis Choices." *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19,* ACM Press, 2019, pp. 1–13. Crossref, doi:10.1145/3290605.3300322.
2. Rule, Adam, et al. "Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding." *Proceedings of the ACM on Human-Computer Interaction, vol. 2,* no. CSCW, Nov. 2018, pp. 1–12. Crossref, doi:10.1145/3274419.
3. Kyle Kelley and Brian Granger. 2017. Jupyter frontends: From the classic Jupyter Notebook to JupyterLab, Interact, and beyond. (2017). Talk. JupyterCon.
4. Codoban, Mihai, et al. "Software History under the Lens: A Study on Why and How Developers Examine It." 2015 *IEEE International Conference on Software Maintenance and Evolution (ICSME),* IEEE, 2015, pp. 1–10. Crossref, doi:10.1109/ICSM.2015.7332446.
5. Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. *In Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP'12)*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation. cfm?id=2342875.2342882
6. João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2016. Fine-Grained Provenance Collection over Scripts Through Program Slicing. *In International Provenance and Annotation Workshop*. Springer, 199–203.
7. Yoon, YoungSeok, and Brad A. Myers. "Supporting Selective Undo in a Code Editor." *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* IEEE, 2015, pp. 223–33. Crossref, doi:10.1109/ICSE.2015.43.
8. Kery, Mary Beth, et al. "Variolite: Supporting Exploratory Programming by Data Scientists." *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17,* ACM Press, 2017, pp. 1265–76. Crossref, doi:10.1145/3025453.3025626.
9. Head, Andrew, et al. "Managing Messes in Computational Notebooks." *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*, ACM Press, 2019, pp. 1–12. Crossref, doi:10.1145/3290605.3300500.
10. Glassman, Elena L., et al. "OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale." *ACM Transactions on Computer-Human Interaction*, vol. 22, no. 2, Mar. 2015, pp. 1–35. Crossref, doi:10.1145/2699751.

11. Glassman, Elena L., et al. “Visualizing API Usage Examples at Scale.” *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI ’18*, ACM Press, 2018, pp. 1–12. Crossref, doi:10.1145/3173574.3174154.
12. Pavel, Amy., et al. “Browsing and Analyzing the Command-Level Structure of Large Collections of Image Manipulation Tutorials”. Technical Report No. UCB/EECS-2013-167, 2013
13. Kong, Nicholas, et al. “Delta: A Tool for Representing and Comparing Workflows.” *Proceedings of the 2012 ACM Annual Conference on Human Factors in Computing Systems - CHI ’12,* ACM Press, 2012, p. 1027. Crossref, doi:10.1145/2207676.2208549.
14. Ko, Andrew J., and Brad A. Myers. “Finding Causes of Program Output with the Java Whyline.” *Proceedings of the 27th International Conference on Human Factors in Computing Systems - CHI 09*, ACM Press, 2009, p. 1569. Crossref, doi:10.1145/1518701.1518942.
15. Burg, Brian, et al. “Explaining Visual Changes in Web Interfaces.” *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST ’15,* ACM Press, 2015, pp. 259–68. Crossref, doi:10.1145/2807442.2807473.
16. Rule, Adam, et al. “Exploration and Explanation in Computational Notebooks*.” Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI ’18*, ACM Press, 2018, pp. 1–12. Crossref, doi:10.1145/3173574.3173606.
17. Srinivasa Ragavan, Sruti, et al. “Foraging Among an Overabundance of Similar Variants.” *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems - CHI ’16,* ACM Press, 2016, pp. 3509–21. Crossref, doi:10.1145/2858036.2858469.
18. Zhang, K. Z., et al. “Approximate Tree Matching in the Presence of Variable Length Don′t Cares.” *Journal of Algorithms,* vol. 16, no. 1, Jan. 1994, pp. 33–66. Crossref, doi:10.1006/jagm.1994.1003.