Interactions for Untangling Messy History in a Computational Notebook

Mary Beth Kery
Human-Computer Interaction Institute, CMU
mkery@cs.cmu.edu

Brad A. Myers
Human-Computer Interaction Institute, CMU
bam@cs.cmu.edu

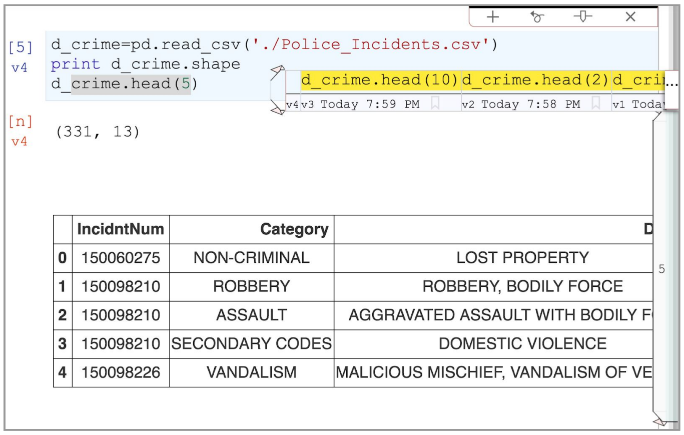
***Abstract*—Experimentation through code is central to data scientists' work. Prior work has identified the need for interaction techniques for quickly exploring multiple versions of the code and the associated outputs. Yet previous approaches that provide history information have been challenging to scale: real use produces a high number of versions of different code and non-code artifacts with dependency relationships and a convoluted mix of different analysis intents. Prior work has found that navigating these records to pick out the *relevant* information for a given task is difficult and time consuming. We introduce Verdant, a new system with a novel versioning model to support fast retrieval and sensemaking of messy version data. Verdant provides light-weight interactions for comparing, replaying, and tracing relationships among many versions of different code and non-code artifacts in the editor. We implemented Verdant into Jupyter Notebooks, and validated the usability of Verdant's interactions through a usability study.**

Keywords—exploratory programming, versioning, data science

I. INTRODUCTION

In data science, exploratory programming is essential to determining which data manipulations yield the best results [1] [2], [3]. It can be highly helpful to record what iterations were run under what conditions and under what assumptions about the data. This gives data scientists better certainty in their work, the ability to reproduce it, and a more effective understanding about where to focus their efforts next. Today, most of this experimental history is lost. Our studies [4] [5], as well as those by Rule et al [6] and a 2015 survey from Jupyter of over 1000 data science users [7] have all found task needs as well as strong direct requests from data scientists for improved version support.

In terms of versioning, where does data science programming diverge from any other form of code development? Typically in regular code development, the primary artifact that a programmer works with is code [8]. Data science programming relies on working with a broader range of artifacts: the code itself, important details within the code [4], parameters or data used to run the code [9], visualizations, tables, and text output from the code, as well as notes the data scientist jots down during their experimentation [10]. The conditions under which code was run and under which data was processed gives meaning to a version of code [9]. Data scientists need to ask questions that require knowledge of history about specific artifacts, specific code snippets, and the relationships among those artifacts over time: "What code on what data produced this graph?", "What was the performance of this model under these assumptions?", "How did this code perform on this dataset versus this other dataset?", etc. Seeing relationships among artifacts allows a data scientist to answer cause-and-effect questions and evaluate the results and the progress of their experimentation.



```
[5]  d_crime=pd.read_csv('./Police_Incidents.csv')
v4   print d_crime.shape
     d_crime.head(5)

[n]
v4    (331, 13)
```

	IncidntNum	Category	D
0	150060275	NON-CRIMINAL	LOST PROPERTY
1	150098210	ROBBERY	ROBBERY, BODILY FORCE
2	150098210	ASSAULT	AGGRAVATED ASSAULT WITH BODILY F
3	150098210	SECONDARY CODES	DOMESTIC VIOLENCE
4	150098226	VANDALISM	MALICIOUS MISCHIEF, VANDALISM OF VE

Figure 1. Verdant in-line history interactions. For the top code cell, a ribbon visualization shows the versions of the third line of code. In the output cell below, a margin indicator on the right shows that there are 5 versions of the output.

To achieve this level of history support, we aim to A) store a rich relational history for all artifacts, B) allow data scientists to pull out history specifically relevant to a given task, and C) clearly communicate how versions of different artifacts have combined together during experimentation.

Several related areas of tooling offer promising avenues towards these goals. Computational notebook development environments, such as Jupyter notebooks, have become highly popular for data science programming because a notebook allows a data scientist to see all their input, output, formatted notes, and code artifacts in one place, and thus more easily work with and communicate context [11]. Meanwhile, our prior research prototype called Variolite demonstrated several in-editor interactions for creating and manipulating versions of specific code snippets [4]. Only working with code snippets,

978-1-5386-4235-1/18/$31.00 ©2018 IEEE

Variolite did not treat the issue of a mix of code and non-code artifacts or issues of scalability, thus further exploration of this form of lightweight in-editor interactions is needed to adapt these ideas to more complex situations. Finally, the field of provenance research, meaning "origin or history of ownership" [12], has argued for and developed methods for automatically collecting input, code, and output each time a programmer runs their code, in order to capture a complete history [13], [14]. Currently the best solutions available to data scientists are manually making Git commits at very frequent intervals, manually making copies of their code files, or manually writing logging code for parameter and output artifacts they want to record [4]. Besides lessening the burden on the programmer to manually version their artifacts, automated approaches can detect and store dependency relationships among artifacts [13].

Unfortunately, just collecting the appropriate history data is not enough. Prior provenance research illustrates that in real use, capturing history data produces a large number of versions with complex dependency relationships and a convoluted mix of different analysis intents that can become overwhelming for a human to interpret [13]. Behavioral research has found that it is both a challenging and tedious task for human programmers to pick out and adapt relevant version data from long logs of code history [15]. Even when using standard version control like Git, software developers often struggle with information overload from many versions, all of which are rarely labeled or organized in a clear enough way to easily navigate [8].

In this work, we explore the design space of new interactions for providing easy-to-use history support for data scientists in their day-to-day tasks. Untangling messy history logs to deliver them in a useful form requires both advances in how edit history is modeled, and active testing of potential user interactions on actual log data from realistic data science tasks. To facilitate this, we developed a prototype tool called *Verdant* (from the meaning "an abundance of growing plants" [16]) as an extension for Jupyter notebooks. By relying on existing Jupyter interactions to display code and non-code artifacts, Verdant adds a layer of history interactions on top of Jupyter's interactions that are likely to be familiar to data scientists and already have been established to be usable even to novices [17]. Underlying Verdant, we develop a novel approach to version collection to model versions of all artifacts in the notebook along with dependency relationships among them. Using this gathered history data, we then explore the design space of lightweight interactions for:

1. Quickly retrieving versions of a specific artifact out of an abundance of versions of the entire document.
2. Comparing multiple versions of different artifacts including code, tables, and images, which benefit from different diff-ing techniques.
3. Walking the data scientist through how to reproduce a specific version of an artifact.

Finally, we validate the real life task-fit of these interactions in an initial usability study with five experienced data science programmers. All participants were successfully able to complete small tasks using the tool and discussed use cases for Verdant specific to their own day-to-day work. With feedback from these use case walkthroughs from participants, we discuss next steps in his design space.

II. Related Work

Computational Notebooks: Computational notebook programming dates back to early ideas of "literate programming" by Knuth [18] in 1984. Although there are many examples today of computational notebooks like Databricks [19] or Colab [20], Jupyter is a highly popular and representative example with millions of users. Therefore, we chose to use it in this work, particularly since it is open-source and thus easy to extend. Computational notebooks show many different artifacts together in-line. Each artifact, like code or markdown, has its own "cell" in the notebook, and the programmer is free to execute individual cells in any order, thus avoiding needing to re-run computationally expensive steps. The cell structure is an important consideration for versioning tools. Since the cell is a discrete structure, it can be tempting to version a notebook by cells so that the user can browse all history specific to one cell. However, we caution against overly relying on cell structure, because prior behavioral work [5], [6] shows that notebook users commonly add lots of new cells, then reduce or recombine them into different cell structures as they iterate. Users also reorder and move around cells [5], [6]. Finally, Jupyter notebooks support "magic" commands, which are commands that start with "`%`" that a user can run in the notebook environment to inspect the environment itself. This includes a `%history` command that outputs a list of all code run in the current session. While prior history work in Jupyter notebooks [13] has relied on `%history`, we take a different approach since this `%history` prints only the plaintext code run in a tangle of different analysis tasks, and we aim to collect more specific context across all artifacts involved.

Provenance work: Provenance, tracking how a result was produced, has many different levels of granularity, all the way down to the operating system-level of the runtime environment [14]. In this work, we do not collect *absolute* provenance, since we only collect reasonably fine-grained runtime information about code, input, and output that is accessible from inside the computational notebook environment. The focus of our research is how to make provenance data *usable* to data scientists, and thus we focus on recording the history metadata most useful for data scientists at the cost of some precision. Pimentel et al. in 2015 created an extension to Jupyter notebooks that collects Abstract Syntax Tree (AST) information to record the execution order and the function calls used to produce a result [13]. However, to retrieve this history, users must write SQL or Prolog queries into their notebook to retrieve either a list of metadata or a graph visualization of the resulting dependencies [13]. Instead of having users write more code to retrieve history, our focus in this work is to provide direct manipulation interactions which require far less skill from the user. Extensive prior provenance work has used graph visualizations to communicate provenance relationships to users [21], however graph visualizations are well known to be difficult for end-users to use [22], thus we avoid them.

Version History Interaction Techniques: In standard code versioning tools like Git, versions are shown as a list of commits, or a tree visualization to show different branches in a series of commits [23]. In tools like R Studio [24] a data scientist can see a list of code they have run so far. However just like Jupyter's `%history` list, a list of code lacks any context to tell which code went with which analysis tasks or artifact context like input/outputs/notes needed to return to a prior version. Variolite tackles more specific version context by structuring in tool form the informal copy-paste versioning that data scientists already use [4]. In Variolite, programmers are able to select a little section of code, even just a line or a parameter, and wrap it in a "variant box" so that within that box, they can switch among multiple versions. Rather than shifting through full versions of the whole file, the programmer has the code variants that are meaningful to them directly in the editor. However, Variolite did not provide any support for non-code artifacts, and was highly limited by the manually drawn variant box. Variolite only recorded snippet-specific history inside the variant box, so the user could not move code in and out of the box without losing history. If a user did not think to put a variant box around everything of interest *before* running code experiments, it was not possible to recover snippet-specific history later. To avoid these limitations, our new Verdant system automatically collects all history so that a data scientist can flexibly inspect different parts of their work and always have access to its history data. Finally, prior work for fine-grained selective undo of code has collected versioning on a token-by-token level and visualized this through in-editor menus and an editor pane displaying a timeline [25]. Token-level edits are not terribly appropriate for data scientists because during experimentation, data scientists are more concerned with semantically-meaningful units of code like a parameter or method, rather than low level syntax edits [3].

Behavioral work on navigating versions: Navigating corpuses of version data and reusing bits of older versions has been shown to be difficult for programmers, from professional software engineers to novices [8], [15]. Srinivasa Ragavan et al. have modeled how programmers navigate through prior versions using Information Foraging Theory (IFT) [15] in which a programmer searches for the information by following clues called "scents". Scents include features of a version like its timestamp, output, and different snippets found within the code. To investigate how data science programmers specifically mentally formulate what aspect of a prior version they are looking for, we ran a brief survey with 45 participants [5]. We found that data scientists recall their work through many aspects like libraries used, visual aspects of graphs, parameters, results, and code, not all of which are easily expressed a textual search query [5]. Given these findings, we aim to support foraging and associative memory by providing plenty of avenues for a data scientist to navigate back to an experiment version based on whatever tidbit or artifact attribute they recall.

III. VERDANT VERSION MODEL

Verdant is built as an Electron app that runs a Jupyter notebook, and is implemented in HTML/CSS and Node.js. Although the interactions of Verdant are language-agnostic, since the implementation relies on parsing and AST models for code versioning, Verdant relies on a language-specific parser. We chose to support Python in this prototype, as it is a popular data science language. By substituting in a different parser, Verdant can work for any language.

Verdant uses existing means in Jupyter for displaying different types of media in order to capture versions of all artifacts in the notebook. For a single version of the notebook, (a "commit" using Git terminology), the notebook is captured in a tree structure. The root node of the tree is the notebook itself, and each cell in the notebook is a child node of the notebook. For code cells, their nodes are broken down further into versions by their abstract syntax tree (AST) structure, such that each syntactically meaningful span of text in the code can be recorded with its own versions. For output, markdown, and other multimedia cells, the cell is a node with no children, which means that a programmer can see versions of the output cell as a whole, but not of pieces of output.

A full version of the notebook is captured each time any cell is run. For efficiency, commits only create new nodes for whatever has changed, using reference pointers to all of the child nodes of the previous commit for whatever is the same. Versioning in this tree structure and at the AST level addresses many concerns of scale. For instance, imagine a data scientist Lucy has iterated on code for 257 different runs, but has only changed a certain parameter 3 times. Through AST versioning, Lucy does not have to sift through all 257 versions of her code with repetitive parameter values, but can instead simply retrieve the 3 unique versions of that parameter. Although AST versioning provides a great deal of flexibility to provide context-specific history, like Lucy's 3 versions of her parameter, it adds algorithmic challenges. Namely, each time Lucy runs her code, there is the full version A of the AST which is the last recorded version of the program and a new full version B of the AST that is the result of all of Lucy's new changes up to the point of the run. Matching two ASTs has been done previously, using heuristics like string-distance, type and tree structure properties [26][27], however note that this matching has not been used in user-facing edit tracking before. Further, what is a correct match from a pure program structure perspective may not always match what is “correct” to the user. For instance, if Lucy changes a parameter `3` to `total(“Main St.”)`, Lucy may want to see the history of these two AST nodes matched, since both are serving the exact same role as her parameter, however since these are far in both type and string-distance, a traditional matching algorithm would *not* match the two. Refining this matching algorithm to match user expectation is an area for future work, thus for the purposes of the immediate design exploration, Verdant implements a simple Levenshtein string matching algorithm: if the token edit distance between two AST nodes is less than or equal to 30% of the length of the nodes, Verdant considers them a match.

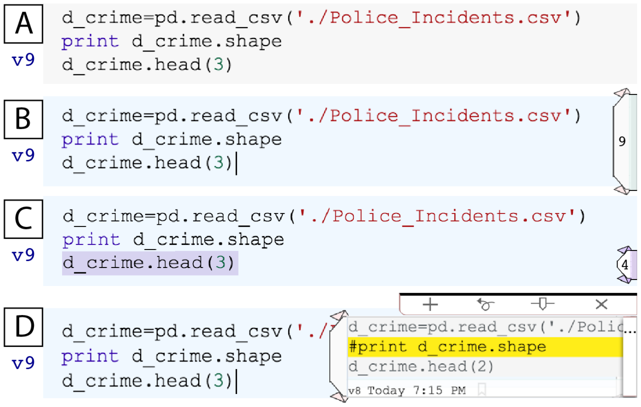
To collect dependency relationships, we run the Jupyter magics command %whos, which returns information from the running python kernel on the names and values of variables currently present in the notebook’s global environment. When one of these global variables changes value, we record which code cell ran immediately before the variable change, to approximate which code cell set the value of that variable, consistent with some prior code execution recording work [28]. For each code node that Verdant versions, Verdant inspects the code’s AST structure to identify which if any of the global variables that code snippet uses. If the code snippet uses a global variable, then a dependency is recorded between the code node and that specific version of the global variable, including which other code version produced the used value of the variable.

V. INTERACTIONS FOR VERSION FORAGING

Although a notebook may contain many code, output, and markdown cells, prior work suggests that data scientist work on only a small region of cells at a time for a particular exploration [5]. First, we show how Verdant uses inline interactions so that users can see versions of the task-related artifacts they are interested in, and not be overloaded with unrelated version information for the rest of the notebook.

A. Ambient Indicators

Following tried and tested [29] usability conventions of other tools that support investigating properties of code, like “linters,” a version tool should be non-disruptive while the user is focused on other tasks, while giving some ambient indication of what information is available to investigate further. Linters often use squiggly lines under code and indicator symbols in the margins next to the line of code the warning references. Verdant takes the approach in Figure 2: (A) no version information appears when a data scientist is reading through their notebook, but (B) when data scientists click on a cell to start working with it, they see an indicator in the margin that gives the number of versions of that cell (in this case 10). If the data scientist selects different spans of code, the indicator changes height and label to show the number of unique versions of the selected code (in this case 9) (C). While a linter conventionally puts an icon on one line, we decided instead for the height of the version indicator to stretch from the bottom to the top of the text span it is referencing to more clearly illustrate which part of the code the information is about. Finally, if the programer clicks on the indicator, this will open the default active view, the ribbon display (D) with buttons for reading and working with the versions of that artifact, as described next.



```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(3)
```

```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(3)
```

```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(3)
```

```
d_crime=pd.read_csv('./
print d_crime.shape
d_crime.head(3)
```

```
d_crime=pd.read_csv('./Polic...
#print d_crime.shape
d_crime.head(2)
v8 Today 7:15 PM
```

Figure 2. (A) no version information shown, (B) on selecting a cell an margin indicator displays how many versions of that artifact there are, (C) on selecting specific code the indicator updates to show how many versions of that specific snippet there are, (D) upon clicking the indicator, a ribbon visualization shows versions of an artifact starting with the most recent

B. Navigating Versions

The “ribbon display” shown in Figure 1D is the default way Verdant shows all versions for an artifact, lined up side by side to the right of the original artifact. Unlike existing code interactions like a linter or autocomplete, where a pop-up may appear in the active text to supply short static information, versioning data is comprised of a long ordered list of information and must continuously update as the data scientist runs their code. So in the ribbon visualization, because code and cells in the notebook are read from top to bottom, the version property of an artifact is visualized left to right, with the leftmost version, which is shown in blue (Figure 2D), always being the *active version*. Here “active version” will always refer to the version of the artifact that is in the notebook interface itself and that is run when the user hits the run button in the notebook. Since the ribbon is a horizontal display, it can be navigated by horizontal scroll, the right and left arrow keys, or by clicking the ellipsis bar at the far right of the ribbon which will open a drop-down menu of all versions (Figure 2D). For navigating versions, note that Variolite made a different design decision, and had users switch between versions via tabs. However, as suggested by Variolite’s usability study [4], as well as recent work on tab interfaces in general [30], tab interfaces do not scale well as the number of versions increases beyond a handful. On the other hand, choosing from a list display is not the most speedy for retrieving an often used version if it is far down on the list. The ribbon display always shows the most *recent* versions first, making recent work fast to retrieve on the intuition that recent work is more likely to be relevant to the user's current task. For non-recent items, bookmarking is a standard interaction for fast retrieval of often-used items. Since robust history tools currently do not exist, we lack grounded data on which history versions a data scientists is likely to use. So to probe this question with real data scientists, we added an inactive bookmark icon to all versions, indicating that the user *can* bookmark it. We use this during our initial usability test to probe potential users on bookmarking, their use cases (if any) for it, and on various ways it could be displayed (see below).

C. Comparing Versions

Among many versions, is is important for a data scientist to quickly pick out what is important about that version out of lots of redundant content. This also helps provide "scents" for users to further forage for information. If a data scientist Lucy opens a cell's version and sees that only a certain line has changed much over the past month, she can adjust the ribbon to show only versions in which that line changed, hiding all other versions that are not relevant to that change.

In Verdant, a diff is shown in the ribbon and timeline views by highlighting different parts of a prior version in bright yellow (Figure 2D) For code, Verdant runs a textual diff algorithm consistent with Git, and for artifacts like tables that are rendered through HTML, Verdant runs a textual diff on the HTML versions and then highlights the differing HTML elements.

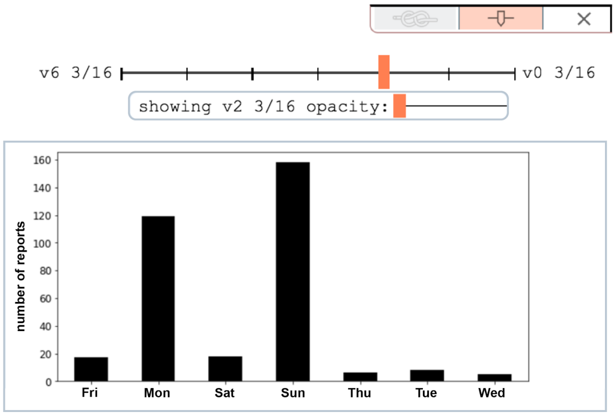


Figure 3. Timeline view. By dragging the top orange bar side to side the user can change the version shown. By dragging the lower orange bar, the user can set the opacity of the historical version they are viewing, in order to see it overlayed on top of the currently active output version.

Since an "artifact" can be a tiny code snippet or a gigantic table or a graph or a large chunk of code, one-size-fits-all is not the best strategy for navigation and visualization across all these different types. For instance, for visual artifacts like tables or images, visualization research [31] has found that side-by-side displays can make it difficult to "spot the difference" between two versions. In the menu bar that appears with the default ribbon, a user can select a different way of viewing their versions. For visual artifacts, overlaying two versions is suggested, so a timeline view can be activated (Figure 3), by selecting the ⇩ symbol. A data scientist can navigate the timeline view by dragging along the timeline, or by using the right/left arrow keys.

For visual diffing, Verdant again relies on advice from visualization research [31] and uses opacity in the timeline view where the user can change the opacity of a version they are looking at to see it overlayed on top of their currently active version.

For all artifact types, there are multiple kinds of comparisons that could be made, each of which optimizes for a different (reasonably possible) task goal:

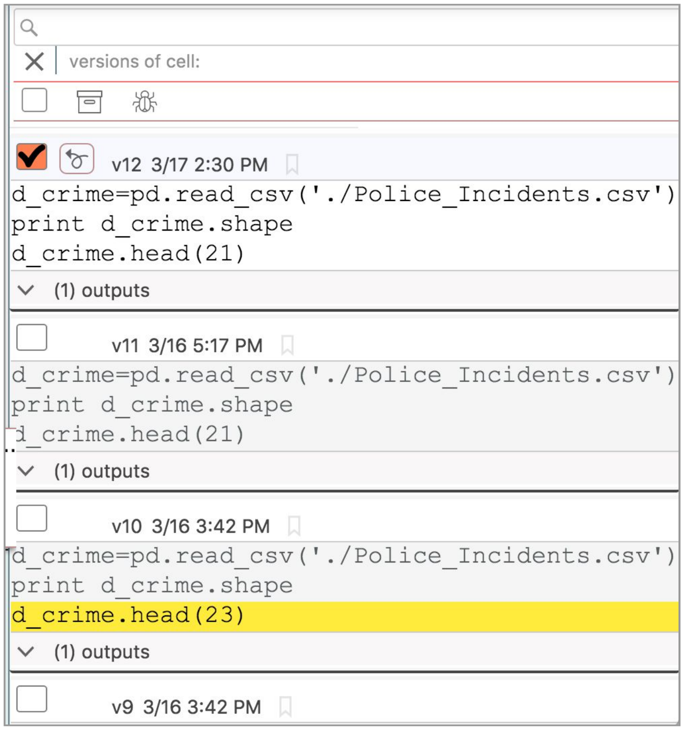
1. What is different between the active version and a given prior version?
2. What changed in version N from the version immediately prior?
3. What changed in version N from version M, where M and N are versions selected by the data scientist from a list of versions?

For an initial prototype of Verdant, we chose to implement the first option only, on the hypothesis that spotting the difference between the data scientist's immediate current task and any given version will be most useful for spotting useful versions of their current task out of a list. We then used usability testing to probe through discussion with data scientists which kinds of diff they expect to see, and what task needs for diffs they find important (see study below).

D. Searching & Navigating a Notebook's Full Past

In-line versioning interactions allow users to quickly retrieve versions of artifacts present in their immediate working notebook, but has the drawback that some versions cannot be retrieved this way. The cell structure of a notebook evolves as a data scientist iterates on their ideas and adds, recombines, and deletes cells as they work [5]. Suppose that Lucy once had a cell in the notebook to plot a certain graph, but later deleted it once that cell was no longer needed. To recover versions of the graph, Lucy cannot use the in-line versioning discussed above, because that artifact no longer exists whatsoever in the notebook: she has no cell to point to and indicate to "show me versions of this". So to navigate to versions not in the present workspace, and to perform searches, Verdant also represents all versions in a list side pane.

The list pane can be opened by the user with a button, and is tightly coupled with the other visualizations such that if the user selects an artifact in the notebook, the pane will update to list all versions of that artifact, and stay consistent with the current selected version. If no artifact is selected, the list shows all versions of the notebook itself. With a view of the entire notebook's history, the user can see a chronologically ordered change list beginning with the most recent changes across all cells in the notebook. Say Lucy wants to retrieve a result she produced last Wednesday that has since been deleted from her current notebook. Either by scrolling down the list or by using the search bar to filter the list by date, she can navigate to versions of her notebook from last Wednesday to try to pull out the relevant artifact when it last existed. Alternatively, she can use the search bar to look for the result by name. Note that Lucy does not need to actually find the *exact* version she is looking for from this list. Using foraging, if she can find the old cell in the list that she thinks at some point produced the result she is thinking about, she can select that cell in the list to pull up all of its versions of code and output. From there, she can narrow her view further to only show the output produced on Wednesday. This method of searching relies on following clues across dates and dependency links among artifact versions, rather than requiring the data scientist to recall precise information that would be needed for a query in a language like Prolog [13].



```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(21)
```

```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(21)
```

```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
d_crime.head(23)
```

Figure 4. In the list view, the user can select one or more versions to act upon. With the search bar, the user can filter versions using keywords or dates.

VI. REMIX, REUSE, & REPRODUCE

When data scientists produce a series of results, they may later be required to recheck how that result was produced. Common scenarios include inspecting the code that was used to check that the result is trustworthy, or reproducing the same analysis on new data [5]. Without history, reproducing results is commonly a tedious manual process, where the data scientist re-creates the original code from memory [5].

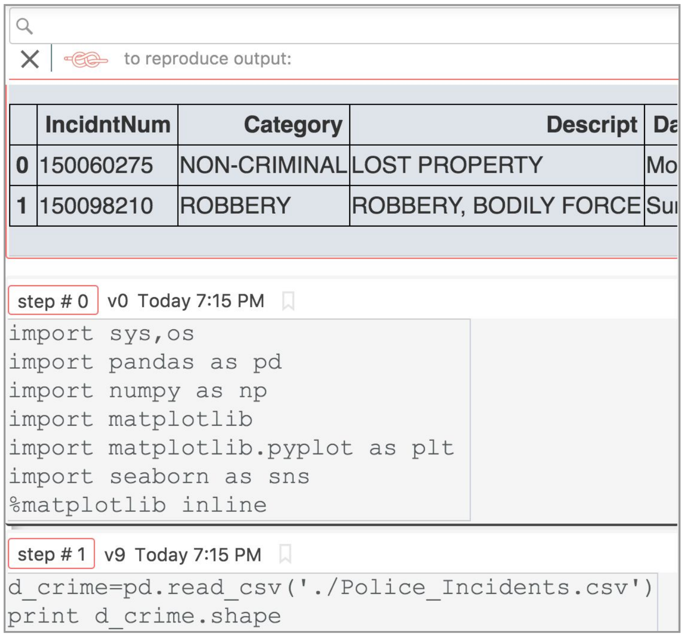
A. Replay older versions

To replay any older version of an artifact in the notebook, a user in Verdant can make that version the *active* version and then re-run their code. In any of the in-line or list visualizations of an artifact's versions, the data scientist can select an older version of an artifact and use the ⟲ symbol button to make that version the active one. The formerly active version for that artifact is not lost, since it is recorded and added as the most recent version in the version list. If a data scientist wants to replay a version of an artifact that no longer exists in the current notebook, that artifact will be added as a new cell of the current notebook, located as close as possible to where it was originally positioned.

Although this interaction can be used to make any older version the active one, it completely ignores dependencies that the older version originally had. Our rationale behind this is clarity and transparency: if Lucy clicks the ⟲ symbol on a certain version, that changes only the artifact the version belongs to. If instead Verdant also updated the rest of the notebook, changing other parts of the notebook to be consistent with the version dependencies, then Lucy may have no understanding of what has changed. In addition, sometimes data scientists use versions more as a few different options for doing a particular thing (e.g., to try a few different ways for computing text-similarity) and are not interested in the last context the code-snippet-version was run in, just in reusing the specific selected code-snippet-version. To work with prior experiment dependencies, Verdant provides a feature called "Recipes", described next.

B. Output Recipes

What code should a data scientist re-run to reproduce a certain output? Once the data scientist finds the output they would like to reproduce, they can use the ⛓ symbol button (shown at the top of Figure 3). Verdant uses the chain of dependency links that it has calculated from the output to produce a *recipe visualization*, shown in Figure 5. The "recipe" appears in the side list pane as an ordered list of versions labeled "step 1" to "step N". Consistent with all other visualizations, the recipe highlights in yellow any code in the steps that is different from the currently showing code in the notebook. So, if a code cell is entirely absent from the notebook, it will be shown in entirely yellow. If a matching code cell already exists in the notebook and perfectly matches the active version, it will be shown in entirely grey in the recipe with a link to navigate to the existing cell to indicate that the data scientist can just run the currently active version of that code. Note this dependency information is imperfect, because we do not version the underlying data files used, so if the dataset itself has changed, the newly produced output may be still different than the old one. We discuss several avenues for versioning these data structures in Future Work.



	IncidntNum	Category	Descript	Da
0	150060275	NON-CRIMINAL	LOST PROPERTY	Mo
1	150098210	ROBBERY	ROBBERY, BODILY FORCE	Su

step # 0 v0 Today 7:15 PM

```
import sys,os
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

step # 1 v9 Today 7:15 PM

```
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape
```

Figure 5. In the recipe view, a user sees the output they selected first, and then an ordered series of code cells that need to be run to recreate that output.

B. *Trust and Relevancy of versions at scale*

A data scientist will try many attempts during their experimentation, many of which may be less successful or flawed paths [4], [32]. Thus, especially when collaborating with others, it can be important for a data scientist to communicate which paths failed [5], [8] and how they got to a certain solution. “Which path failed” requires a kind of storytelling that it unlikely automated methods can capture, thus it would be most accurate for the data scientist to label certain key versions themselves. However, we know from how software engineers use commits (often lacking clear organization or naming) that programmers can be reluctant to spend any time on organizing or meaningfully labeling their history data [8]. Under what circumstances would data scientists be motivated to label trustworthiness and relevancy of their code? To experiment with interactions for this purpose, Verdant uses an interaction metaphor of email in the version list. Like in email, the data scientist can select one or many of their versions from the list (filtering by date or other properties through the search bar) and can “archive” these versions so that they are not shown by default in the version list. Also, the data scientist can mark the versions as “buggy” to more strictly hide the versions and label them as artifacts that contain dangerous or poor code that should not be used. If an item is archived or marked buggy, it still exists in the full list view of versions (so that it can be reopened at any time), but it is hidden by being collapsed. If an item is archived or marked buggy and has direct output, those outputs will be automatically archived or marked buggy as well. A benefit of using a familiar metaphor like archiving email for a prototype system is that it is much easier to communicate the intent of this feature to users. During the usability study, discussed below, we showed the archive and “buggy” buttons to data scientists to probe how, if, and under what tasks they would actively manually tag versions like this.

VII. Initial Usability Test

Verdant is a prototype that introduces multiple novel types of history interaction in a computational notebook editor. Thus it is necessary to test both the usability of these interactions and also to investigate through interview probes how well these interactions seem to or meet real use cases to validate that our designs are on the right track.

For our study setup, we aimed to create semi-realistic data analysis tasks and history data. For Verdant to store and show data science history at scale and in realistic use, we anticipate a later stage field study where data scientists would work on their own analysis tasks in the tool over days and weeks. Here for an initial study, we avoid participants having to work extensively on creating analysis code by instead asking them to use Verdant to try to *navigate* and *comprehend* the history of a fictitious collaborator’s notebook. To create realism, we chose this notebook out of an online repository of community-created Jupyter notebooks that are curated for quality by the Jupyter project [33]. From this repository we searched for notebooks that contained very simple exploratory analyses and that needed no domain-specific knowledge to ensure the notebook content would not be a learning barrier to participants. The notebook we chose does basic visualizations of police report data from San Francisco [34]. Since currently detailed history data is not available for notebooks, we edited and ran different variations of the San Fransisco notebook code ourselves to generate a semi-realistic exploration history.

Next, we recruited individuals who A) had data science programming experience, B) were familiar with Python, and C) had at least two months experience working with Jupyter notebooks. This resulted in five graduate student participants (1 female, 4 male) with an average of 12 years of programming experience, an average of 6 years of experience working with data, and an average of 3 years experience using notebooks. In a series of small tasks, participants were asked to navigate to different versions of different code, table, and plot artifacts using the ribbon visualization, diffs, and timeline visualization. The study lasted from 30 to 50 minutes and participants were compensated $20 for their time. Participants will be referred to as P1 to P5.

All participants were able to successfully complete the tasks, suggesting at least a basic level of usability. Among even a small sample, we were surprised by the diverse use cases participants expressed that they had for the tool. P1 and P5 expressed that they would like to use the ribbon visualization of their versions about every 1-2 days to reflect on their experiment’s progress or backtrack to a prior version. P2 was largely uninterested in viewing version history, but instead was enthusiastic about using the ribbon visualization to switch between 2 to 4 different variants of an idea. P3 was less interested in viewing version history of code cells, but greatly valued the ability to view and compare the version history of output cells. P3 commonly ran models that took a long time to compute (so they only wanted to run a certain version once), and currently to compare visual outputs, had to scroll up and down their notebook. However, P3 did appreciate the ability to version a code cell, as a safe way of keeping their former work in case they wanted to backtrack later. Finally P4 primarily used notebooks in their classwork, and were very enthusiastic about using code artifact history to debug, revert to prior versions, and to communicate to a teaching assistant what methods they had attempted so far when they went to ask for help. P2, P3, and P4 expressed they would like to use the inline history visualizations "all the time" when doing a specific kind of task they were interested in, whether that be comparing outputs or code.

In this initial study, a participant's imagined use case affected which features of Verdant they cared about most. When probed about the use of bookmarking, P2 felt strongly that bookmarks would be useful for their use case of switching among a few different alternative versions, however the other participants who had a more history-based use case were neutral about bookmarking. For the probe in which we showed email-like buttons for archiving or marking code as buggy, participants had very divergent opinions. P1 said they would want to mark versions as buggy and said that they would want to group a bunch of versions and leave a note about what the problem was, but would never use the archive functionality. P2 said they would likely mark versions as buggy, but would be wary of using the archive button to hide older or unsuccessful content. P2 disliked the "archive" metaphor because they felt the relevance of different versions was too task dependent: a version that seems worth archiving in one task context might be very relevant for a future task. All other participants were neutral about the two options, and saw themselves using them to curate their work occasionally. While participants said they would use the inline visualizations daily or every other day when working within a notebook, they said they would use the list pane or recipe visualizations only once a week or once a month. P5 said that although they imagined themselves tracing an output's dependency rarely, this feature was extremely valuable to them when needed, since currently when P5 must recreate output, this was a tedious and error-prone manual process of trying to re-code its dependencies from memory.

In terms of diffing, all five participants were familiar with and used Git, and all guessed that the yellow-highlighted diff in Verdant, like Git, showed what had changed from one version to the next. When we clarified that yellow highlighting showed the diff between any version and the *active* version of the artifact, two participants said that was actually more helpful for them to pick which other versions to work with. All participants wanted the option of multiple kinds of diff. P3, who primarily wanted to diff output, asked for more kinds of visual diffing than the timeline scroll such as setting opacity to see two versions overlayed (which we added into the current Verdant) and a yellow-highlighting for image diffs. Finally, multiple participants disliked horizontal scroll for navigating the ribbon visualization (horizontal scroll is not a gesture on many mice devices) and prefered the ribbon's dropdown menu to select versions.

VIII. FUTURE WORK & CONCLUSION

There remain many key directions in supporting experiment history. Our small user study revealed a high variance of an individual's day-to-day task needs for their history. Thus to truly understand the impact of Verdant and future tools in this space, a key next step is to conduct a field study across a larger number of participants over several weeks in order to collect grounded data on how data scientists put history to use in practice. There remain many further systems design directions as well, particularly to visualize differences and comparisons between different kinds of artifacts. While we demonstrate Verdant on images and code, different visualizations may be more useful and effective to portray the history of tables, plots, or notes. Future work is also needed to help collect version information about data files, to ensure reproducibility without consuming too much memory space. In work such as Verdant, we move from considering code history only for engineering practice to building human-centered history tools for experts and end-user programmers to synthesize their ideas, and more responsibly conduct experimentation and exploration.

ACKNOWLEDGMENTS

We thank our pilot participants. This research was supported in part by a grant from Bloomberg L.P., and in part by NSF grant IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funders.

REFERENCES

[1] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, "Interactions with big data analytics," *Interactions*, vol. 19, no. 3, pp. 50–59, 2012.

[2] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer, "Enterprise Data Analysis and Visualization: An Interview Study," *IEEE Trans. Vis. Comput. Graph.*, vol. 18, no. 12, pp. 2917–2926, Dec. 2012.

[3] P. J. Guo, "Software tools to facilitate research programming," Doctor of Philosophy, Stanford University, 2012.

[4] M. B. Kery, A. Horvath, and B. A. Myers, "Variolite: Supporting Exploratory Programming by Data Scientists," in *ACM CHI Conference on Human Factors in Computing Systems*, 2017, pp. 1265–1276.

[5] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool," in *ACM CHI Conference on Human Factors in Computing*

Systems, 2018, p. 174.

[6] A. Rule, A. Tabard, and J. Hollan, “Exploration and Explanation in Computational Notebooks,” in *ACM CHI Conference on Human Factors in Computing Systems*, 2018, p. 32.

[7] “Jupyter Notebook 2015 UX Survey Results,” *Jupyter Project Github Repository*, 12/2015. [Online]. Available: https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb.

[8] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, “Software history under the lens: a study on why and how developers examine it,” in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, 2015, pp. 1–10.

[9] K. Patel, “Lowering the barrier to applying machine learning,” in *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology*, 2010, pp. 355–358.

[10] A. Tabard, W. E. Mackay, and E. Eastmond, “From Individual to Collaborative: The Evolution of Prism, a Hybrid Laboratory Notebook,” in *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, San Diego, CA, USA, 2008, pp. 569–578.

[11] F. Pérez and B. E. Granger, “IPython: a System for Interactive Scientific Computing,” *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, May 2007.

[12] “provenance - Wiktionary.” [Online]. Available: https://en.wiktionary.org/wiki/provenance. [Accessed: 22-Apr-2018].

[13] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire, “Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow,” in *Workshop on the Theory and Practice of Provenance (TaPP), Edinburgh, Scotland*, 2015, pp. 155–167.

[14] P. J. Guo and M. I. Seltzer, “Burrito: Wrapping your lab notebook in computational infrastructure,” in *4th UNSENIX Workshop on Theory and Practice of Provenance*, 2012.

[15] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, “Foraging Among an Overabundance of Similar Variants,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, San Jose, California, USA, 2016, pp. 3509–3521.

[16] “verdant - Wiktionary.” [Online]. Available: https://en.wiktionary.org/wiki/verdant. [Accessed: 22-Apr-2018].

[17] R. J. Brunner and E. J. Kim, “Teaching Data Science,” *Procedia Comput. Sci.*, vol. 80, pp. 1947–1956, Jan. 2016.

[18] D. E. Knuth, “Literate programming,” *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.

[19] “Databricks,” 2013. [Online]. Available: https://databricks.com/.

[20] “Colaboratory,” 2018. [Online]. Available: https://colab.research.google.com/. [Accessed: 22-Apr-2018].

[21] K. Cheung and J. Hunter, “Provenance explorer--customized provenance views using semantic inferencing,” in *International Semantic Web Conference*, 2006, pp. 215–227.

[22] I. Herman, G. Melançon, and M. S. Marshall, “Graph visualization and navigation in information visualization: A survey,” *IEEE Trans. Vis. Comput. Graph.*, vol. 6, no. 1, pp. 24–43, 2000.

[23] S. Chacon and B. Straub, “Git and Other Systems,” in *Pro Git*, S. Chacon and B. Straub, Eds. Berkeley, CA: Apress, 2014, pp. 307–356.

[24] R. Team and Others, “RStudio: integrated development for R,” *RStudio, Inc. , Boston, MA URL http://www. rstudio. com*, 2015.

[25] Y. Yoon, B. A. Myers, and S. Koo, “Visualization of fine-grained code change history,” in *2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2013, pp. 119–126.

[26] I. Neamtiu, J. S. Foster, and M. Hicks, “Understanding source code evolution using abstract syntax tree matching,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[27] R. Koschke, R. Falke, and P. Frenzel, “Clone Detection Using Abstract Syntax Suffix Trees,” in *2006 13th Working Conference on Reverse Engineering*, 2006.

[28] S. Oney and B. Myers, “FireCrystal: Understanding interactive behaviors in dynamic web pages,” in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2009, pp. 105–108.

[29] “IntelliJ IDEA,” *IntelliJ IDEA*. [Online]. Available: https://www.jetbrains.com/idea/. [Accessed: Apr-2017].

[30] Nathan Hahn, Joseph Chee Chang, Aniket Kittur, “Bento Browser: Complex Mobile Search Without Tabs,” in *2018 CHI Conference on Human Factors in Computing Systems*, 2018, p. 251.

[31] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts, “Visual comparison for information visualization,” *Information Visualization; Thousand Oaks*, vol. 10, no. 4, pp. 289–309, Oct. 2011.

[32] K. Patel, J. Fogarty, J. A. Landay, and B. Harrison, “Investigating statistical machine learning as a tool for software development,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 667–676.

[33] “A gallery of interesting Jupyter Notebooks.” [Online]. https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks. [Accessed: 24-Apr-2018].

[34] lmart, “SF GIS CRIME,” *GitHub*. [Online]. Available: https://github.com/lmart999/GIS. [Accessed: 27-Apr-2018]