Towards Effective Foraging by Data Scientists to Find Past Analysis Choices

Mary Beth Kery
Human-Computer Interaction Institute, Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
mkery@cs.cmu.edu

Bonnie E. John, Patrick O'Flaherty
Bloomberg L.P.
731 Lexington Ave
New York, New York 10022
bjohn11@bloomberg.net, poflaherty2@bloomberg.net

Amber Horvath, Brad A. Myers
Human-Computer Interaction Institute, Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
ahorvath@cs.cmu.edu, bam@cs.cmu.edu

ABSTRACT

Data scientists are responsible for the analysis decisions they make, but it is hard for them to track the process by which they achieved a result. Even when data scientists keep logs, it is onerous to make sense of the resulting large number of history records full of overlapping variants of code, output, plots, etc. We developed algorithmic and visualization techniques for notebook code environments to help data scientists forage for information in their history. To test these interventions, we conducted a think-aloud evaluation with 15 data scientists, where participants were asked to find specific information from the history of another person's data science project. The participants succeed on a median of 80% of the tasks they performed. The quantitative results suggest promising aspects of our design, while qualitative results motivated a number of design improvements. The resulting system, called Verdant, is released as an open-source extension for JupyterLab.

CCS CONCEPTS

•Human-centered computing → Human computer interaction (HCI); •Software and its engineering → *Software creation and management;*

KEYWORDS

Literate Programming; Exploratory Programming; Data Science; End-User Programmers (EUP), End-User Software Engineering (EUSE)

ACM Reference format:
Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, and Amber Horvath, Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of ACM SIGCHI, Glasgow, UK, May 2019 (CHI'19),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Data analysis and machine learning models have an increasingly broad and serious impact on our society. Yet "data-driven" does not actually imply effective, correct or benevolent unless the humans creating these models are able to effectively reason about the analysis choices they make. As part of the large legal and research push for analyses and models to be explainable, a data scientist must be accountable for their analysis choices [7]. It may help the data scientists be more aware and productive if, just as natural scientists keep a lab notebook of their experiments, they too had support to quickly record and reference what they already tried, under what assumptions, and with what results [27, 31]. Unfortunately the typical process of data science is a series of highly exploratory and winding small code experiments, making it very difficult in practice for a data scientist to achieve a tidy overview of progress [15, 17].

Despite limited current support, many savvy data scientists do take notes and use version control tools (e.g. Git) to record their work [15, 17]. Another common strategy is to copy all scripts, outputs, or even full computational notebooks to establish checkpoints [15, 17]. However, if a data scientist wants to answer a concrete question from their prior work, such as *"why did I discard this data feature from my model?"*, they need more support, including that:

Req. 1) History is sufficiently complete: the experiments that led to each particular choice must have been recorded in the first place. Ideally the history should keep all relevant artifacts needed for a user to understand an experiment (in case reproduction is not easily feasible) including all the plots, code, tables, notes, data, etc. that were used [16].

Req. 2) History is reasonably conveyed for comprehension, so that the cost of tracking down an answer is not prohibitive.

Prior studies show that data scientists do not typically save their history at frequent-enough intervals to capture all their experimentation [15]. To address this requirement, recent tools [2–4, 16] now help provide a complete or semi-complete history by automatically capturing checkpoints of a data scientist's work at regular intervals, such as every time that users run their code.

One important barrier against requirement 2 is that data science experimentation quickly generates a large number of versions that can be too dense from which to draw information. For instance, when the first author worked on coding a beginner tutorial machine learning problem, within about 1 hour, the code had been edited and run 302 times. Lists of versions are highly susceptible to the long repetitive list problem [29]. Essentially, if there is a long list of similar variants of the same document, it is a laborious process for the user to search through them [29]. For pure code questions, a Git expert user may be able to use Git `bisect` or `blame` to track

CHI'19, Glasgow, UK

DOI: 10.475/123_4

down where specific code changed. However for visual artifacts like plots or fuzzier questions like *"why did I discard this data feature"*, the user is pushed into a tedious brute force search, reviewing version after version until they find the information they need. As a participant from [17] put it: *"it's just a lot of stuff and stuff and stuff."* If answering a quick historical question would take a disproportionately long time, a data scientist will not do it [17].

Previously, we built a Jupyter-Notebook-based code editor, Verdant [16], in which we prototyped interactions that a data scientist might perform with their own history: activities like reproducing outputs and comparing or viewing versions of artifacts. In this paper, we investigate support for the specific challenges that data scientists face around question-answering from history. First, we extended Verdant to serve as a base history recording infrastructure for new history-searching designs. We significantly refined the history recording and retrieval mechanisms to improve efficiency (described below). To allow our designs to be easily available to other researchers and data scientists, we freshly re-implemented Verdant to work as an extension for JupyterLab, a publicly available development environment for Jupyter Notebooks. We then outfitted Verdant with conventional search interactions, like a search bar for history and filters designed specifically for the context of data science activity.

With infrastructure to make history search possible, how do we help data scientists *effectively* answer questions from their history? Prior work from *code foraging theory* [14, 24, 29] has studied how programmers find useful information from source code. Drawing from foraging research on how to signal useful information out a long list of versions, Verdant provides foraging cues like date, version previews, and `diff` highlighting to show the meaningful ways that various versions of artifacts differ [29]. The current release of Verdant, presented here, includes the following contributions:

- Spatial visualizations of notebook activity over time, and techniques for interacting with them.
- Inspector interactions, analogous to the web browser style inspector for CSS, that allow a user to click on artifacts of interest to "inspect" the history specific to that artifact.
- A new kind of notebook document, which we call a "ghost book," which allows the user to compare full past notebook versions with their current notebook.
- A refinement of our previous history model [16] with significant performance improvements. We release this model as lilGit, an open-source extension for JupyterLab.

Verdant provides all of these features, and is an open-source extension to the also open-source JupyterLab so that both researchers and the data science community can easily acquire and extend these designs[1]. Finally, we conducted an evaluation of Verdant using realistic tasks. This study showed that 15 data scientists using Verdant for the first time were able to correctly answer a median of 80% of the tasks they were given in a data science project that was completely new to them comprised of over 300 versions.

2 BACKGROUND & RELATED WORK

Data science involves yielding insights, creating models, visualizing phenomena, and many other tasks that make use of data. Although data scientists work with a variety of tools, including spreadsheet editors and graphical visualization tools, writing programs using languages like Python and R is prevalent in data science due to their power and the wide availability of reusable open-source resources like IPython [23] and Scikit-learn [21]. Working with data is a need that spans almost all sectors of industry and science, meaning that a "data scientist" can be anyone from an engineer to a chemist to a financial analyst, to a student [12]. In our research we focus specifically on the creation and prototyping parts of a data science workflow (as opposed to maintenance or dissemination [12]), and thus focus our tool design work on computational notebooks, which are widely used by millions of data scientists for this purpose [1, 18].

2.1 Foraging in source code

Information foraging theory (IFT), developed by Pirolli and Card [25], stems from the biological science concept of optimal foraging theory as applied to how humans hunt for information. IFT includes certain constructs adopted from optimal foraging theory: *predators* correspond to humans who are hunting for information, their *prey*. They perform these hunts in parts of the UI, called *patches*. In the context of foraging in software engineering, the programmer is the predator, the patch is an artifact of the environment which can vary from a single line of code to a generated output or a list of search results, and the piece of information that the programmer is looking for is the prey. A *cue* is the aspect of something on the user's screen that suggests a particular place that they should look next.

IFT has been applied to source code foraging in a variety of domains including requirements tracing, debugging, integrated development environment (IDE) design, and code maintenance [11, 19, 20, 22, 24]. The design of our tool builds upon this work by taking into account design implications for how programmers forage for information [12, 19, 22] by providing specific foraging cues such as dates, previews, and `diff` highlighting. We apply this theory to a new set of users, data scientists, and base our experiment design on prior foraging experiment designs [29].

2.2 Version control and collaboration

Version control and collaboration tools for data science programming are a growing focus of both research and industry. Although "data science" is a relatively new term, the practice of exploratory programming with data is long established [12] and prior work has found that data scientists underutilize traditional versioning tools like Git [15]. Collaboration in data science tasks is made more difficult by the number of code and non-code artifacts involved in experimentation, which are onerous to diff or merge in a traditional code versioning system like Git [1].

In recent work, Google's Colaboratory project [5] avoids this software engineering flavor of versioning altogether by providing a notebook environment in which multiple collaborators can simultaneously edit a computational notebook, much like a Google Doc [6]. Although a gain for real-time collaborative data science, this is a different focus from our current research, where we concentrate on helping data scientists understand past experimentation.

Research projects like Variolite [15], our prior version of Verdant [16], Azurite [32], and ModelTracker [8] have all focused on helping

[1]Verdant: https://github.com/mkery/Verdant

programmers track their exploratory work. The distinction of our current work is that we are focused on foraging and finding.

3 DESIGN USE CASE OVERVIEW

Given the breadth of data science tasks, we first analyzed available data on specific questions data scientists have articulated that they want to understand from their history [17]. We used these data to map out use cases to guide our design:

(1) A data scientist is working alone with their final results as the deliverable. Over a long period of work, they use history as a memory aid to check their intermediary results.
(2) A data scientist is communicating their in-progress experimentation to a colleague. For instance, an analyst is using history to justify a model to her boss.
(3) History is sent along with a data science notebook for process transparency. For instance, a professor can use history to understand how a student got to a specific result on an assignment.

For now, the collaborative examples above still assume history is coming from a single data scientist. Given the new interaction space and the still understudied area of collaborative data science, we argue starting with exploring how an individual data scientist can navigate history is a important first step.

4 DESIGN FOR VERSIONING ARTIFACTS

For the current release of Verdant, we created a version model, called *lilGit*, based on Git [9], the near-ubiquitous version control tool for software engineering. Each file in a directory for which Git is responsible is called a *"blob"*, and each blob has its own history via a series of file-copies stored in the hidden `.git` directory. What this means in practice is that a software developer can quickly access their history at two levels of granularity: the list of commits that constitute the history of the entire project, or just the versions specific to when a particular file was changed. The fundamental assumption that Git makes is that the software developer's artifacts of interest are their code files and the commit messages. However, this assumption breaks down for data scientists — in a computational notebook, for instance, the document is filled with crucial artifacts like snippets of code, formatted "markdown" text around the code, and visual output like charts and graphs that constitute individual analyses. To answer a fine-grained question about a code snippet may be done with Git using various combinations of Git `blame`, `grep`, `pickaxe`, and `log`, all of which have their drawbacks, such as producing many duplicate versions (`grep`) or not accounting for a snippet moving around in the file (`blame`). None of these commands are particularly easy to use, and typically fail on any kind of non-textual artifact, like a plot or chart. Thus, to give data scientists the same level of convenience that software engineers experience with Git, lilGit builds upon our prior model in [16] and Git to work at a fine-grained level.

Artifacts in lilGit are a hierarchical grammar shown in Fig. 1, that breaks down a computational notebook artifact into smaller and smaller artifacts, down to low-level code snippet artifacts, such as a single parameter value. For code artifacts, we rely on an Abstract Syntax Tree (AST) representation of the code to form the hierarchy. Just like each blob in Git has its own history, each artifact holds

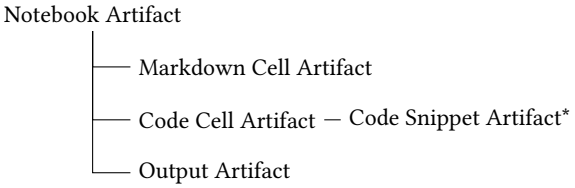

Figure 1: Artifacts types in lilGit. The notebook artifact is a singleton for the entire document. Each cell has it's own artifact and Code cells are further broken down into code snippets. Code snippets correspond to the abstract syntax tree (AST) structure of their parent code cell. Thus they have types, like *function declaration* or *Boolean* and can have many child snippet artifacts according to the AST structure.

Notebook saved	Notebook loaded		
Cell run	Cell deleted	Cell added	Cell moved

Figure 2: Events are JupyterLab UI actions that lilGit listens to. Received events trigger lilGit to update its history data.

a list of its own history comprised of a mix of raw visual/textual data and pointers to versions of child artifacts. For instance, to recreate the state of a code cell artifact at a certain point in time, lilGit would re-create the exact code from all the code cell's child artifacts. This hierarchical approach prevents most duplicates so that a user can easily access unique versions of any artifact.

This tree history structure is is saved in a single `JSON` file called `foo.ipyhistory` which sits next to the user's Jupyter notebook file `foo.ipynotebook`. The benefit of history in a single file is that it is easily portable: a data scientist can choose to share their notebook either with or without their history file.

4.1 Versioning procedure

Step 1. **Notebook is loaded.** Open the notebook's `.ipyhistory` if it exists, and check to see if the last recorded version of the notebook matches the current notebook. If not, use the resolve algorithm (steps 4-6) to create or update the history model.
Step 2. **User makes an edit.** Pick the most specific possible artifact that the user edited and mark it with a ★. This marks the artifact as *potentially* changed.
Step 3. **Notebook-level event.** An event such as a run or save (all listed in Fig. 2) occurs, which triggers the save of a new version to begin in steps 4-6.
Step 4. **Resolve.** For each artifact that is marked with a ★, estimate whether it has changed using a simple textual equals:
  (a) If no change, remove the ★, which will remove the artifact from further consideration.
  (b) Otherwise:
    (i) **Generate** the new artifact entry in the history. If the artifact is code, process the new code through a custom parser that uses Python 3's built-in AST module to generate a new artifact tree.

(ii) **Match** the new artifact against the old one. For code, this again requires program analysis using features like type, position in the AST, and string distance to estimate the pointers between the old code artifact tree to the new one. Any child-artifacts that the matching decides are either changed or new are marked with a ★.

Step 5. **Commit.** Starting from the leaves of the artifact tree for the entire notebook, all artifacts marked with a ★ have a new version permanently recorded. Next, the parents of those nodes, traversing up the tree to the notebook artifact, have new versions committed to account for the new changes in their children. Finally all ★ markers are removed.

Step 6. **Save to file.** Write the new model to the `.ipyhistory` file as the latest version of the user's work.

With this process running in the background of the user's notebook session, Verdant's user-facing interfaces receive updates from this history model to display the user's history, as discussed next.

5 DESIGNING FOR IMPROVED FORAGING [2]

Three tabs top the current design of the Verdant sidebar (Fig. 3 at A, B, C), each supporting a different foraging strategy users can employ to answer their questions.

First, the Activity tab (open in Fig. 3) visualizes history shown by time and event so that the user can forage based on their memory on *when* and *where* a change occurred. A temporal representation of history is core to many other history tools like a conventional undo pane or a list of commits in Git. Second, the Artifacts tab organizes history per artifact so that a user can forage based on *what* artifact changed and *how* a certain artifact evolved over time. Third, the search tab offers a structured search through text queries and filters, which is useful when the users have a search keyword in mind or when their memories of when or where to start looking for an answer to their question are less precise. Each interface is next described in detail.

[2]Verdant's UI has evolved through many design iterations. The latest design is shown in the figures and discussed in the text unless otherwise noted.

5.1 When? Where? Foraging in the Activity tab

Consider a use case where a data scientist has been iterating for a few hours on code for a regression, and asks *"what were the beta values from the regression I ran earlier today?"* [17]. Each artifact version in Verdant is tied to a global-level *event* that triggered it, e.g., a run or save of the notebook (Fig. 2). These are displayed in the Activity tab as a chronologically ordered stream of events (Fig. 3) so that the user can visually scan down to the rough date and time that constitutes "earlier today".

A second global level of referencing time is the versions of the notebook artifact (shown as #55, #54... in Fig. 3). If each event were to have its own row in the stream, the user would need to scroll a long way to get a notion of what had occurred within just a few minutes. To give the visualization a bit denser information yield for foraging, all events that share the same notebook version are chunked into the same row (e.g., #53 at Fig. 3, E). Additionally, run events that occur in non-overlapping cells within in the same 60 seconds are recorded onto the same notebook version. This slightly reduces the granularity of notebook versions, but allows the user to see activity at a glance by minute, rather than by seconds.

Minute by minute may serve to spot recent activity, but a data scientist looking for *"earlier today"* will likely not recall the exact minute something occurred. However, a user might know where in

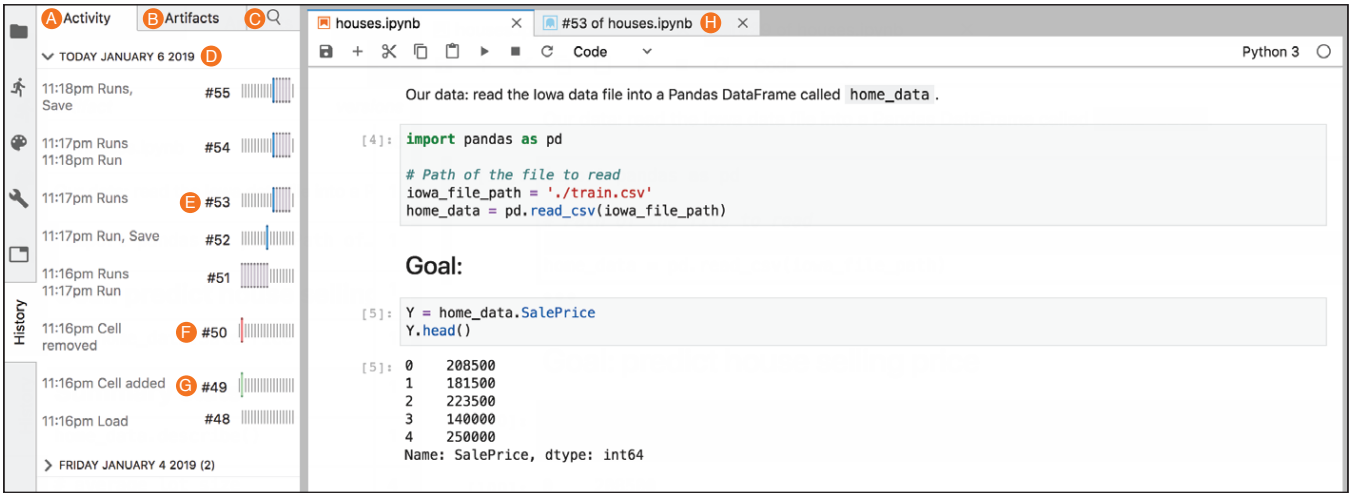

Figure 3: The history tab opens the sidebar for Verdant containing three tabs: Activity (A), Artifacts (B & Fig. 5), and Search (C & Fig. 7). The Activity tab, shown open here, displays a list of events. A date (D) can be opened or collapsed to see what happened that day. Each row shows a version of the notebook (e.g. version #53) with a text description and visual minimap. The minimap shows cells added in green (see G) and deleted in red (F). In (E), a cell was edited and run (in blue), and the following cells were run but remained the same (in grey). The user can open any version (e.g., #53, H & Fig. 8) in a ghost notebook tab for quick reference.

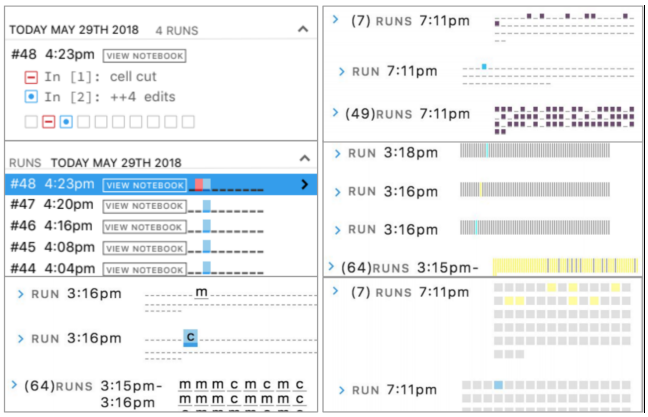

Figure 4: Six design explorations of color, shape, and information content to summarize notebook activity. The final design is shown in Fig. 3

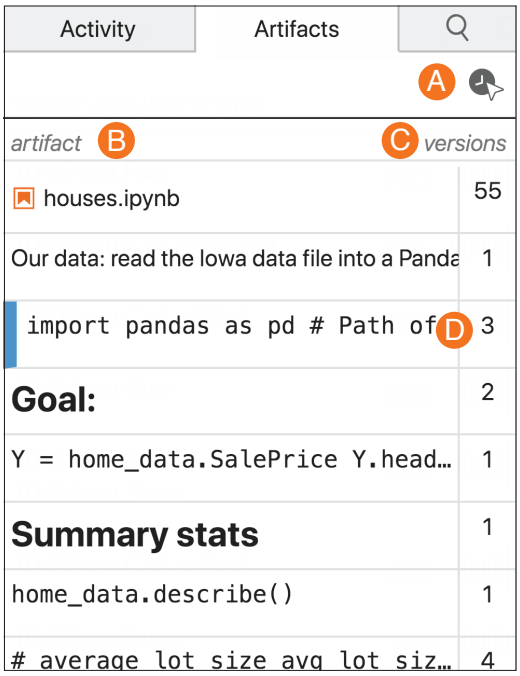

Figure 5: The Artifacts tab's table of contents view shows a summary table. To the left (B) is a preview of the notebook and each cell in the notebook. To the right (C) is the number of versions that artifact has. For instance, the 2nd cell (D) has 3 versions. Using the inspector button (A), the user can select any artifact from their notebook, including code snippets and output not summarized here, to see the detail view in Fig. 6.

the notebook they were working. Perhaps the answer lies during a time when many cells were added to the end of the notebook, or during a time when several cells in the middle were being edited and consolidated. We explored many designs to succinctly visualize *where* in the notebook activity occurred, so that a user may rely on spatial familiarity with their own work to visually rule out entire series of activity where irrelevant areas of the notebook were edited (Fig. 4). Although it might be tempting to display textual labels, as in the top left of Fig. 4, cells in a Jupyter Notebook are (currently) anonymous. The bracketed numbers to the left of cells in Jupyter notebooks (Fig. 3) are not stable and change as cells are added, deleted, or moved over time. To overcome these problems with names and to provide a tighter visualization, we were inspired by both a kind of tiny inline plot popularized by Tufte, called a sparkline [30], and a variation on a common code editor navigation visualization called a minimap[3]. A conventional code minimap shows a miniature shape of the code file with colorful syntax highlighting so that a user can click based on spatial memory of where something occurs in their file, rather than reading exact lines. Prior work has suggested that notebook navigation limits the typical maximum number of cells in people's notebooks to roughly 60 [17, 26] and so we explored various aspects of shape, color, textual content, etc., to summarize key information at a glance, that would smoothly scale up to 60 cells (Fig. 8). In Verdant's final minimap design, the notebook is flipped counter-clockwise to show the series of cells horizontally to conserve space. Each series of vertical lines after the notebook version number represents the entire notebook at that point in time. Each vertical line represents a cell and a taller bold line indicates activity: blue for cell edits, green for cell creation, red for cell deletion, and grey for running a cell without editing it. This representation makes it easy to spot such common cues as where cells have been added, or which portion of the notebook has undergone substantial editing.

5.2 What? How? Foraging in the Artifacts tab

Consider the case where the artifact is still in the current document, but has been changed since the older version the data scientist is looking for. Like preceding systems Variolite [15], Juxtapose [13] and the first version of Verdant [16], we assume that allowing a user to directly manipulate the artifact in question is the fastest way for them to start foraging for an alternative version of that artifact. In Fig. 5, the Artifacts tab summarizes each cell artifact of the notebook using a single line, along with the number of versions it has had, for a quick way to see the cell histories, much like a table of contents. However, it may well be that a user is interested in a finer-grain artifact, such as the relationship between a certain parameter's values and an output. Another complication is that code snippet artifacts and output artifacts can move from one cell to the next if the user reorganizes the notebook, such that the full history of a code snippet artifact might not be contained in the history of just one cell. To address these challenges, we look to a design pattern from another context where there is a rich relational document full of sub-components each with its own sets of properties: a web page. With a browser's style *inspector*, a developer can simply point to any element or sub-element on the active web page, and

[3]The exact origin of code minimaps is unclear, but most modern code editors have a plugin for it, e.g. Atom's minimap package https://atom.io/packages/minimap.

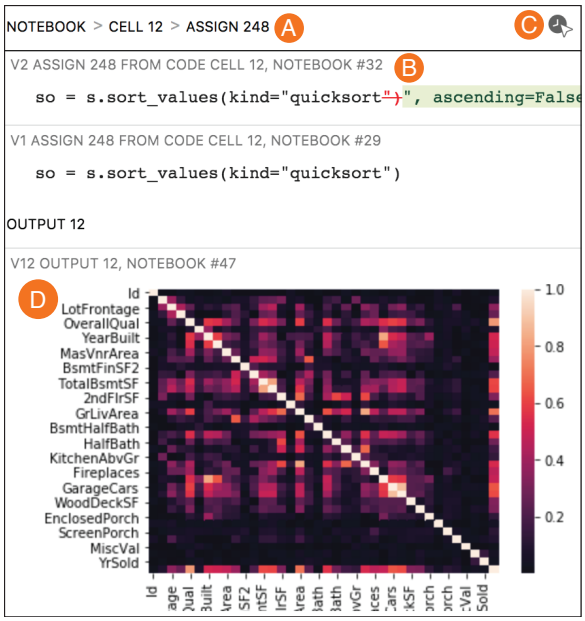

Figure 6: The Artifacts tab showing versions of an assign statement within a code cell (A). Each version is labeled with a unique version # and the notebook version it came from (B). Below the code versions are shown all versions of output (D) that were produced at the same time as those code versions. The user can use the inspector button (C) to select a different artifacts in their current notebook, which will switch the detail view to the selected artifact.

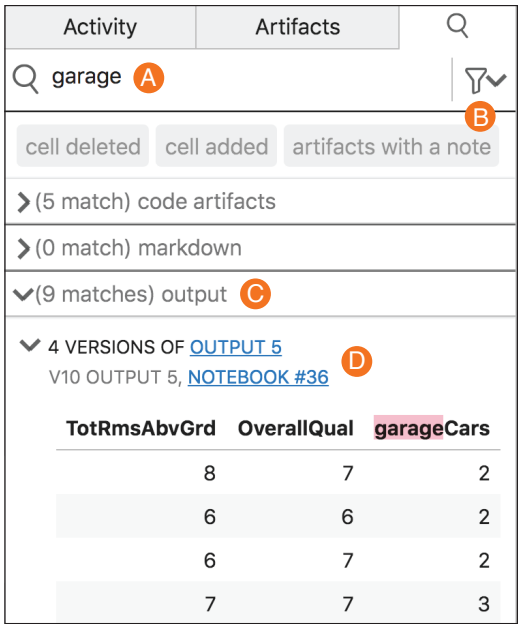

Figure 7: Searching for "garage" (A). The user can use optional filters (B). The results are categorized by artifact type (C). Each match is further organized by artifact. Here, 4 of the 9 matches are versions of a table Output 5 (D).

a browser pane then displays its style properties. This *inspector interaction* is tried and tested across all modern web browsers. We mimic this with a history inspector button (Fig. 5, A) that allows a user to point to any artifact in their notebook. Once a user clicks on an artifact using either the table of contents (Fig. 5, D) or the inspector interaction, Verdant provides a list of unique versions of that artifact (Fig. 6).

5.3 Searching with cues in the Search tab

Imagine that a data scientist is looking for all tables that included a column named "Garage" generated within the last week. If that output is no longer in the notebook, the user will not be able to point to it in the Artifact tab. The Search tab is meant to give users a start when foraging for elements no longer contained in the notebook by searching backwards through the history [32]. By searching for "garage" (Fig. 7), the user receives a list of the matching versions of artifacts. We explored showing all results from all artifact types sorted chronologically, but this led to a glut of information for the user to scroll through, and did not perform well in the evaluation (below). Thus, the Search results are now chunked by artifact type and by artifact ID (Fig. 7) to lower the amount of reading and scrolling required.

5.4 Resurrecting full revisions for context

Although our design criteria for the history tabs in Verdant was to boil down information into small pieces for quick reference, more extended context is needed to answer some questions. If a data scientist wants to ask *"what data was used to generate this plot?"*, the code importing the data and how it was transformed to generate that plot may be spread across multiple locations in the notebook. Although using the Artifacts tab, the user can view the detailed history of any artifact of cell/output size or smaller, we provide a different UI for notebook artifact versions, called a *ghost* notebook. This view allows the user to visualize a prior full notebook, and also shows the context of how specific smaller artifact versions are related to each other in that notebook. As shown in Fig. 8, the ghost notebook is immutable, highlights where changes were made in that notebook version (Fig. 8, D), and has a different background color from the user's active Jupyter notebook to avoid accidentally confusing the two. The two notebooks can be viewed side-by-side, allowing the user to compare the older ghost notebook to their current notebook. The user can also open multiple ghost notebooks to compare across multiple historical states. An example use case for this would be to compare versions of a code file side by side [10] to figure out *"what changed?"* between an earlier working version of the notebook and one that contains a bug.

In addition, the ghost book has a toggle (Fig. 8, C) to show or hide cells unaffected by the edits and runs in this version. This allows users to hide the vast majority of cells in a long notebook and focus their attention on the differences in this ghost book. Note that cells are still marked as *affected* when they are run and compute a value different than the previous execution of that cell, even if their code

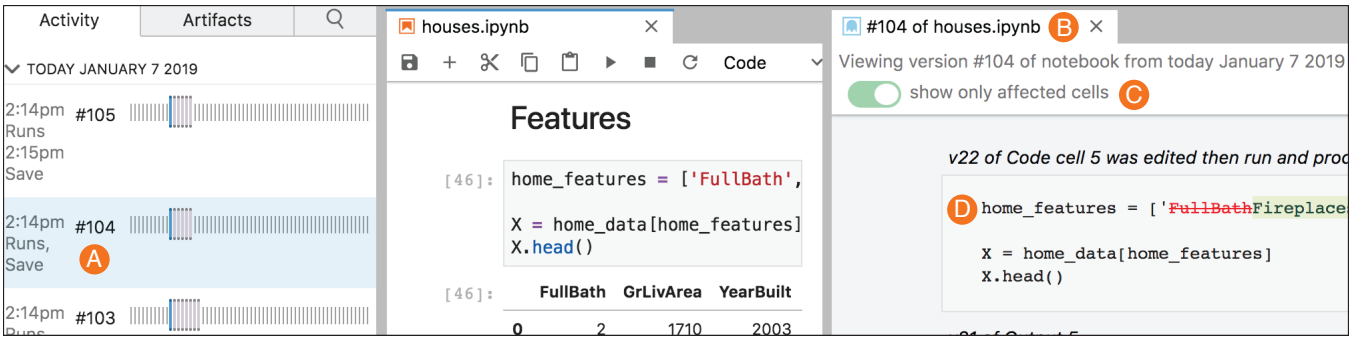

Figure 8: This notebook has about 60 cells. The user clicks a row (A) for version #104 of the notebook in the activity tab, and a Ghost Notebook (B) opens that shows the full details of the notebook at version #104. To make it easier to see the changes, the Notebook has a toggle button to show all cells, or only cells that were run or affected during that version (C). At (D), conventional diff notation shows characters that were deleted in red, and in green characters that were added in that version.

was not edited. For example, editing one cell to change a variable's value might dramatically change a graph output produced by a subsequent cell even if the subsequent cell's code was not edited.

6 EVALUATION OF VERDANT

The primary goal of our evaluation was to gather data about how the features of Verdant assist or hinder data scientists in performing realistic foraging tasks. We had received positive feedback about our ideas from data scientists throughout the design process, so in this evaluation we sought task-based behavioral data to confirm or refute those positive opinions, and provide guidance for redesign. As Verdant is an extension to JupyterLab, we coordinated closely with the Jupyter Project and ran our study at their JupyterCon2018 conference. JupyterCon annually gathers a concentrated group of data scientists, from a variety of sectors, with experience in computational notebooks, providing an opportunity to collect data from professionals with a range of experience in a short period of time.

6.1 Challenges and Limitations of the Study

A conference setting presents considerable challenges to testing a complex tool intended for long-term use by expert users on their own code. Table 1 lays out these challenges and our approach to addressing them.

A major difference between the primary use of a history tool, i.e., querying previous versions of your own code, and what we can study at a conference, is that we had to ask participants to find things in another person's code. Examining other people's code does happen in the real world, e.g., a manager of data scientists told us that he would find Verdant useful for understanding his employees' thought processes, and professors sometimes grade student code on the basis of the process they employed as well as the end-product. Another difference is the skill with the tool itself that a data scientist would build up through long-term use. Both of these problems could be overcome through a longitudinal study or studying professional data scientists after they had used Verdant for several months, which we hope to do in the future.

However, we believe the lack of skill with the tool, no knowledge of the code, and limited time to do the tasks can bring into stark relief any shortcomings in Verdant's UI design. The problems and virtues of Verdant's UI uncovered through performing tasks here give us a glimpse of how useful Verdant would be at least for novice users and what we would need to do to improve it.

Table 1: Constraints of Testing in a Conference Setting

Data scientists in the real world...	Conference attendees...	Study design to address limitations
Work on code for hours, days, or weeks.	Have a maximum of 30 minutes between conference events.	Following [22], we created a substantial notebook. Participants were not asked to write code.
Can take weeks to become skilled with the features of an advanced tool.	Have no prior experience with the tool.	We created a short tour of the tool's features.
Would use tool for their own code, aided by their own memories of their work.	Have no prior exposure to the code we presented to them.	We based the code on beginner tutorials in a simple domain (house sale data).
Have complex, idiosyncratic questions and understand results specific to their own work.	Have no knowledge of what questions are important for this code. Cannot create questions or necessarily recognize the answers.	We based questions on prior data [12], substituting explicit goals for what an author would be able to recognize, e.g., a picture of the chart to find.

6.2 Materials

6.2.1 The Evaluated Verdant JupyterLab Extension. We tested a version of Verdant, which we will call the "evaluated version", which was revised from the version described in [16], and prior to the current version described and pictured above. The current version resulted from redesigns based on the data collected with the evaluated version and the differences will be discussed in the qualitative analysis section below.

6.2.2 The Notebook. In order to create a realistic data science notebook history that both contained substantial experimentation and was simple enough for most participants to understand in a few minutes, we looked at some of the many data science tutorial notebooks available on the web. The first author created a notebook from scratch by following Kaggle's machine learning tutorial level 1 on a housing selling-price dataset, followed by copying in and trying out code from Kaggle community notebooks from a competition with the same dataset[4]. Creating a notebook this way, relying heavily on a variety of other programmers' code, was intended to reduce any bias that the study notebook history would be too specific to one programmer's particular coding style. The resulting 20 cell notebook contained over 300 versions.

6.2.3 The Tour. We wrote eight pages overviewing the tool's features and how they worked. Each page contained a screenshot and annotation that drew attention to a feature and explained how it worked. It took less than 3 minutes to read through this document, and participants could refer back to it at any time.

6.2.4 The Tasks. In a prior needs-elicitation survey at JupyterCon 2017, we asked data scientists *"Given your own work practices, type as many [questions] as you can think of that could be helpful to you to retrieve a past experiment"* [17]. We converted some representative questions from the data scientists into tasks for the current study. Since the participants did not write the notebook, we had to substitute explicit goals for the memories a notebook author would have when setting foraging goals. For instance: *"how did I generate plot 5"* became a task *"Find a notebook version that generated a plot that looks exactly like this [image shown]"* and *"What data sources have I been using over the last month?"* became a task *"How many different data files has this notebook been run on?"*. We generated 15 tasks across 4 task categories (Table 2).

6.3 Participants

JupyterCon2018 provided a User Experience (UX) "Test Fest" room where four organizations set up testing sessions and advertised its availability in the conference program, as slides in meeting rooms between sessions, by some presenters in their talks, and on social media. We recruited 16 participants who came to the UX room (referred to as P1 to P16). Due to equipment issues that arose during P11's session, P11's data will not be considered for analysis, leaving 15 participants. As shown in Table 3, participants performed data science work across a wide range of domains.

Although all participants reported programming experience, one participant reported never having used Python, and one other participant had never used a computational notebook tool, although by attending Jupytercon they would have been exposed to such tools in the presentations. Overall we argue that this is a fairly representative sample of data scientists, except for gender (14 male, 1 female). Two participants had time constraints that interrupted the study, so one attempted 5 tasks, another just 2 tasks, and the remaining 13 participants attempted 6 tasks each.

[4]Tutorial: https://www.kaggle.com/learn/machine-learning, and competition: https://www.kaggle.com/c/house-prices-advanced-regression-techniques.

Table 2: Tasks and number of each task category used

Category	#	Example
Notebook event	3	*Find the first version of the notebook*
Visual finding	3	*Find a notebook version that generated a plot that looks exactly like this [image]*
Code finding	3	*Find the code the author used to check for duplicate houses*
Relation between multiple artifacts	6	*What was the lowest mean absolute error achieved when the author used a RandomForestRegressor?*

Table 3: Data science domain of participants. Some participants have multiple domains.

Computational Domain	Participants
GIS or Earth science	P1, P8
Economics or Finance	P2, P9, P12, P16
Healthcare	P3, P4
Biology, Chemistry, or Physics	P4, P15
HCI or Computer Science	P5, P14
Social Science	P10
Not reported	P6, P7, P13

6.4 Procedure

When participants came to the UX room, they first filled out an online demographic survey. A greeter asked how much time they had to spend testing and, if they had at least 1/2 hour, they were told that a prototype of a JuptyerLab history tool was available to test (among several other types of available activities). When they chose this activity, they were shown to our station and were seated in front of a 27" display, with a keyboard and mouse. They were first given the on-line Tour document to read. They were then given tasks, one at a time, written on index cards and asked to think aloud while working. The order of tasks was randomized across participants using Latin square prior to the study. Screen-capture software recorded the screen and an audio recording was made of their utterances as they worked. As they completed each task, they were given the next card, until they ran out of time and had to go back to the conference sessions. Participants completed no more than 6 tasks each, but all tasks and categories had coverage.

7 QUANTITATIVE ANALYSIS

With an audio and screen recording of all sessions, the first author first reviewed the recordings to note whether a participant had

succeeded or failed each task, based on an answer key. During this process, the authors eliminated two tasks from analysis: Task K (1 participant) became infeasible during the experiment due to a bug in our prototype. The wording of Task H (5 participants) was ambiguous and various participants interpreted it differently. With the remaining 13 tasks, there were 80 foraging instances across the 15 participants.

We used success rate as an indication of how well Verdant supported the users in accomplishing tasks. The average success rate of the participants was 76% (median = 80%), which puts the evaluated version of Verdant close to the average task success rate of usability tests across many domains [28]. Table 4 shows that more than half the participants succeeded at greater than 80% of the tasks they attempted and 20% succeeded at all of their tasks. Despite being asked to answer questions about a substantial notebook they did not write, having to forage through over 300 versions of that notebook, and having no experience with this complex tool, the majority of participants succeeded on the majority of tasks they attempted. For comparison, data scientists interviewed in [17] reported making many local copies of their notebook files. Imagine giving our participants over 300 files and asking them to answer a series of detailed questions about them. Many participants would have run out of time or given up. Even if our participants used Git, as discussed above, they would have had to learn complex command-line search tools and tasks involving graphic output may have been simply impossible. Thus we consider a median 80% success rate to be evidence that the design of Verdant has promise but could be improved.

Table 4: Participant overall success rate

Success rate range	Number of Participants
100%	3
80%-99%	6
67%-79%	4
33%	2

At this stage of development the overall success rate is interesting, but the differential success rate between tasks is more important for further design as it helps us focus on which tasks are more problematic for users. Turning to task success by task category (Table 5). the most difficult kind of task, "relationship between two artifacts", which required hunting down and then relating versions of two or more separate artifacts, had the lowest success rate at 66%. Otherwise, there was no clear relationship between specific tasks we had *a priori* considered to be more "easy" or "complex" based on the number of steps required to accomplish the task. For instance, the tasks at which participants had 100% success were task N: *"How many different data files e.g., 'data.csv' has this notebook been run on?"* (easy, at 3 steps) and task I: *"What was home_features equal to when the mean_absolute_error was below 20,000?"* (complex, at 12 steps).

8 QUALITATIVE ANALYSIS

We turn now to a qualitative usability analysis that investigates which features of the evaluated Verdant UI were helpful and which may have hindered participants in accomplishing their tasks.

To analyze the think-aloud usability data, we first determined the most efficient method to do each task and the UI features that were involved in those methods. We then watched the videos and noted when participants followed or deviated from those methods, as well as positive and negative comments about the features, and suggestions that the participants made. We used the differential success rates discussed above to focus our attention on the tasks with the lowest completion rate.

The data provided information at many levels, from comments on the tour, to buggy behavior, to complaints about low-level UI features like labels or icons that users found inscrutable, to issues with the high-level functionality. As an example of the latter, a data scientist in the Healthcare industry (P4), was concerned that Verdant saved outputs, saying *"We avoid ever checking data into a version control thing. If it was always saving the output, we wouldn't be able to use it."* We will use all this information in future development of Verdant, but for the purpose of this paper, we focus on three problems: confusion about how to navigate within Verdant, the need for excessive scrolling, and participants resorting to brute-force looking through ghost books.

For the tasks with the lowest success rate, O and G, participants would often click something and not know how to get back to where they had been. One third of our participants articulated the equivalent of *"How do I get back?"* in these two tasks alone (P1, P5, P9, P12, P16). Looking more broadly, more than half of the participants (8/15) articulated this problem across 9 of the 15 tasks, with many more clicking around trying to get back without explicitly voicing their need.

To illustrate the scrolling problem, in Task F, the participants had to find a particular heatmap. The heatmap had been added sometime during the 300 versions, had been changed several times (the desired answer being one of those versions), then deleted. Of the 6 participants attempting this task, 5 immediately selected the correct feature (then called the *Run Save tab*) and the correct action (text search). P9 succeeded in 6 seconds because he had performed a graphic search task before and knew to keep scrolling through the results. Four others succeeded within 3 minutes, performing actions in addition to the most efficient method (all tried ghost books; 2 tried the Inspector Tab, which is equivalent to the current Artifact Tab) and those actions provide clues to better design. Consistent with Information Foraging Theory [25], these detours suggest that

Table 5: Success by task category

Task category	# attempted	mean success
Notebook event (A, B, C)	21	78%
Visual finding (F, L)	10	79%
Code finding (D, J, N)	17	81%
Relation between two artifacts (E, G, I, M, O)	30	66%

having to scroll too long before finding promising results causes people lose confidence in the information patch and abandon it.

At a higher level, we observed many participants resorting to a brute-force search. *"It's obvious if I looked at all of these [ghost books], then I'd know the answer, but there's got to be a smarter way to do this."* (P6) They opened up one ghost book at a time until they reached the solution or became so frustrated they switched their foraging tactic (such as searching with a different term) [24] or else quit the task altogether: *"I found 22 things... I can find it, but I'm not sure I have the patience."* (P3). One participant (P10) to our surprise, sat for a full 6 minutes and read through 39 different ghost books before reaching an answer. Although none of the tasks actually required using brute-force search of ghost books, it is a problem that users got to a point where they thought brute-force was the only solution available to them.

These three problems, together with other evidence too numerous to include here, inspired us to redesign the evaluated Verdant to reduce the need to switch tabs, scroll, or open many irrelevant ghost books. The evaluated Verdant had two text search fields, one in each of the tabs, each with slightly different behavior. We redesigned the current Verdant to have a separate Search tab that combined the functionality of the two individual searches (Figure 3, C). This reduced the need to switch between tabs to see the different search results. Further, more visible filters (Figure 7-B) helps users focus on the types of cells they are looking for, as do the collapsible category sections (Figure 7, C). These sections keep relevant results together and minimize the need to scroll or open many ghost books. The current design of the tabs (Figure 3, A, B, C) hopefully will make it easier for users to know how to return to previous views.

9 CONCLUSIONS & FUTURE WORK

This paper presents a novel system to help data scientists examine the history of their work. Our design and evaluation focused on finding specific artifacts, since this is the first step in understanding what and why things were done, and retrieving discarded work if warranted. Efficiently and automatically recording events ensures that the code is always there to be found, removing the requirement that data scientists take the effort to checkpoint each change as they do their experimentation. Using Information Foraging Theory [25] and specifically, code foraging theory [14, 24, 29], as inspiration when designing the UI yielded a tool with which data scientists could succeed at a median of 80% of realistic tasks in querying a large notebook.

Our evaluation collected task-based behavioral data as well as opinions and suggestions from professional data scientists. As future work, we have a trove of bugs to fix, UI elements to tweak, and more areas to redesign than we could present here. We will take all this information into account in future development of Verdant.

While data from people who have only known Verdant for a few minutes is valuable, since Verdant is a complex tool that people need time to learn and the base-case for history tools is people using them on their own code, we plan to deploy the current Verdant and perform long-term studies. In addition, we will deploy Verdant for general use, and welcome interested parties to use it and provide feedback as is common with all JupyterLab extensions.

Some issues that emerged from the evaluation are particularly interesting for future research. First, although we addressed the issue of navigating among the different features of Verdant, this may be a more pernicious problem than our current design can solve. We plan to do a broader design exploration to see if we can integrate the functionality more fully and smooth the transition through different search and filter strategies.

Second, participants had difficulty understanding the flow among the changes in one cell or one version, and how changes ripple through to later versions. This flow brings history understanding into the realm of narrative. Good narrative smoothly ties a series of events together with the key causal and context information needed to make sense of it. We plan to explore the possibility of automatically creating narratives to communicate changes that are separated in both space (different cells) and time (different versions).

The iterative design process being used to create Verdant has proven very effective at identifying the requirements and barriers for data scientists in exploring the history of the exploratory programming that goes into computational notebooks. The features in Verdant are a promising approach to effectively navigating that history. We hope that better communicating a history of experimentation will scaffold data science programmers to follow better practices and more effectively experiment.

ACKNOWLEDGMENTS

The authors would like to thank our participants and the Jupyter project. This research was funding in part through a grant from Bloomberg L.P., and in part by NSF grant IIS-1827385. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] 2015. Jupyter Notebook 2015 UX Survey Results. https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb. (2015).
[2] 2018. Databricks - Making Big Data Simple. https://databricks.com/. (2018). Accessed: 2018-9-20.
[3] 2018. Domino Data Science Platform. https://www.dominodatalab.com/. (2018). Accessed: 2018-9-20.
[4] 2018. Gigantum. https://gigantum.com/. (2018). Accessed: 2018-9-20.
[5] 2018. Google Colaboratory. https://colab.research.google.com/notebooks/welcome.ipynb. (2018). Accessed: 2018-9-20.
[6] 2018. Google Docs - create and edit documents online, for free. https://www.google.com/docs/about/. (2018). Accessed: 2018-9-20.
[7] Ashraf Abdul, Jo Vermeulen, Danding Wang, Brian Y Lim, and Mohan Kankanhalli. 2018. Trends and Trajectories for Explainable, Accountable and Intelligible Systems: An HCI Research Agenda. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 582:1–582:18.
[8] Saleema Amershi, Max Chickering, Steven M Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 337–346.
[9] Scott Chacon and Ben Straub. 2014. *Pro Git*. Apress.
[10] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. 2015. 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*. IEEE, 1–10.
[11] Scott D Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Trans. Softw. Eng. Methodol.* 22, 2 (2013), 1–41.

[12] Philip Jia Guo. 2012. *Software tools to facilitate research programming*. Ph.D. Dissertation. Stanford University.

[13] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. 2008. Design As Exploration: Creating Interface Alternatives Through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*. ACM, New York, NY, USA, 91–100.

[14] Austin Z Henley and Scott D Fleming. 2014. The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2511–2520.

[15] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *CHI*. 1265–1276.

[16] Mary Beth Kery and Brad A Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 147–155.

[17] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. (2018).

[18] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, and Others. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows. In *ELPUB*. 87–90.

[19] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. 2007. Scents in Programs:Does Information Foraging Theory Apply to Program Maintenance?. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*.

[20] Joseph Lawrance, Rachel Bellamy, Margaret Burnett, and Kyle Rector. 2008. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*.

[21] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, Oct (2011), 2825–2830.

[22] Alexandre Perez and Rui Abreu. 2014. A diagnosis-based approach to software comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*.

[23] Fernando Pérez and Brian E Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29.

[24] D Piorkowski, S D Fleming, C Scaffidi, M Burnett, I Kwan, A Z Henley, J Macbeth, C Hill, and A Horvath. 2015. To fix or to learn? How production bias affects developers' information foraging during debugging. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 11–20.

[25] Peter Pirolli. 2007. Information Foraging Theory. In *Information Foraging Theory*. 3–29.

[26] Adam Rule, Aurélien Tabard, and James Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *ACM CHI Conference on Human Factors in Computing Systems*.

[27] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. Ten simple rules for reproducible computational research. *PLoS Comput. Biol.* 9, 10 (2013), e1003285.

[28] Jeff Sauro. 2011. What is a Good Task-Completion Rate? https://measuringu.com/task-completion/. (2011). Accessed: 2019-1-4.

[29] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging Among an Overabundance of Similar Variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 3509–3521.

[30] Edward R Tufte. 2006. *Beautiful evidence*. Vol. 1. Graphics Press Cheshire, CT.

[31] Greg Wilson, D A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven H D Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, Ben Waugh, Ethan P White, and Paul Wilson. 2014. Best practices for scientific computing. *PLoS Biol.* 12, 1 (Jan. 2014), e1001745.

[32] Y Yoon, B A Myers, and S Koo. 2013. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. 119–126.