Managing Messes in Computational Notebooks

Anonymized for submission

ABSTRACT

Data analysts use computational notebooks to write code for analyzing and visualizing data. Notebooks help analysts iteratively write analysis code by letting them interleave code with output, and selectively execute cells. However, as analysis progresses, analysts leave behind old code and outputs, and overwrite important code, producing cluttered and inconsistent notebooks. This paper introduces *code gathering tools*, extensions to computational notebooks that help analysts find, clean, recover, and compare versions of code in cluttered, inconsistent notebooks. The tools archive all versions of code outputs, allowing analysts to review these versions and recover the subsets of code that produced them. These subsets can serve as succinct summaries of analysis activity or starting points for new analyses. In a qualitative usability study, 12 professional analysts found the tools useful for cleaning notebooks and writing analysis code, and discovered new ways to use them, like generating personal documentation and lightweight versioning.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**;

KEYWORDS

Computational notebooks, messes, clutter, inconsistency, exploratory programming, code history, program slicing

ACM Reference Format:
Anonymized for submission. 2019. Managing Messes in Computational Notebooks. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '19).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Data analysts often engage in "exploratory programming" as they write and refine code to understand unfamiliar data, test hypotheses, and build models [19]. For this activity, they frequently use computational notebooks, which supplement the rapid iteration of an interpreted programming language with the ability to edit code in place, and see computational results interleaved with the code. A notebook's flexibility is also a downside, leading to messy code: in recent studies, analysts have called their code "ad hoc," "experimental," and "throw-away" [16], and described their notebooks as "messy" [20, 29], containing "ugly code" and "dirty tricks" in need of "cleaning" and "polishing" [29].

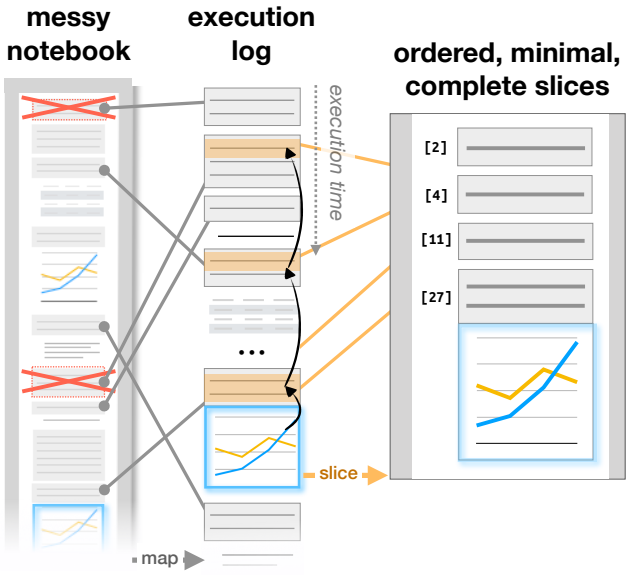

Figure 1: Code gathering tools help analysts manage programming messes in computational notebooks. The tools map selected results (e.g., outputs, charts, tables) in a notebook to the ordered, minimal subsets or "slices" of code that produced them. With these slices, the tools help analysts clean their notebooks, browse versions of results, and discover provenance of results.

In essence, a notebook's user interface is a collection of code editors, called "cells." At any time, the user can submit code from any cell to a hidden interpreter session. This design leads to three types of messes common to notebooks: disorder, where the interpreter runs code in a different order than it is presented in the cells; deletion, where the user deletes or overwrites the contents of a cell, but the interpreter retains the effect of the cell's code; and dispersal, where the code that generates a result is spread across many distant cells. Such messes make it difficult for an analyst to understand the code, and to recall how results (i.e. charts, tables, and other code outputs) were produced, impeding their ability to reuse, extend, and share their analysis code [20, 29].

In this paper, we aim to improve the state of the art in tools for managing messes in notebooks. We introduce a suite of interactive tools, code gathering tools, as an extension to computational notebooks. The tools afford analysts the ability to find, clean, and compare versions of code in messy notebooks. They build on a static program analysis

CHI '19, May 2019, Glasgow, UK
2019. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

technique called program slicing [33], which answers queries about the dependencies among a program's variables. With code gathering tools, an analyst first selects a set of analysis results, which can be any cell output (e.g., charts, tables, console output) or variable definition (e.g., data tables, models). Then the tool searches the execution log—an ordered history of all cells executed—to find an ordered, minimal subset or "slice" of code needed to compute the selected results.

This paper makes two contributions. The first contribution is the design and implementation of *code gathering tools*. These tools offer interactive affordances for managing messes in notebooks. Specifically, the tools highlight dependencies used to compute results, to help analysts find code they wish to understand, reuse, and rewrite in cluttered notebooks. They provide ordered, minimal code slices that can serve as succinct summaries of analysis activity or starting points for branching analyses. Additionally, they archive past versions of results and allow analysts to explore these versions, and the code slices that produced them. Code gathering tools are implemented as an extension to Jupyter Notebook [2], a popular notebook with millions of users [17]. The extension is available for use as a design artifact and as practical tool for exploratory data analysis in notebooks.[1]

The most important idea behind the interaction design of code gathering tools is *post-hoc mess management*—that tools should allow analysts to easily find, clean, and compare versions of code in notebooks, regardless of whether they have followed a disciplined strategy to organize and version their code. Past tools for cleaning code often require effort: annotating cells with dependency information [3], folding and unfolding cells [27], marking and tagging lightweight versions of snippets [18], etc. With code gathering tools, history is stored silently, and tailored slices of code are recalled on-demand with two or fewer clicks.

Our second contribution is a qualitative usability study providing insight into the uses and usability of code gathering tools for managing exploratory messes. 12 professional data analysts used the tools in an in-lab study to clean notebooks and perform exploratory data analysis. We found that affordances for gathering code to a notebook were both valued and versatile, enabling analysts to clean notebooks for multiple audiences, generate personal reference material, and perform just-in-time branching. We also refined our understanding of the meaning of "cleaning," and how code gathering tools supports an important yet still incomplete set of tasks analysts consider as part of code cleaning. This study confirmed that analysts thirst for tools that help them manage exploratory messes, and that code gathering tools provide a useful tool for managing these messes.

[1] To protect the anonymity of this paper, the extension will be made available at the conclusion of blind review.

2 BACKGROUND AND RELATED WORK

Messes in computational notebooks

Lackluster code quality appears to be intrinsic to exploratory programming. Analysts regularly prioritize the speedy discovery of solutions over writing high-quality code [19]. They clutter their programs by saving old versions of their code in comments [18, 35]. In notebooks in particular, poor code quality takes on a spatial dimension. Messes accrue and disappear in an iterative process of expansion and reduction of code: analysts write code in many small cells to try out different approaches to solve a problem, view intermediary output from their code, and debug their code; and then combine and eliminate cells as their analyses reach completion [20].

Eventually, messes get in the way of data analysis. It becomes difficult to understand analyses separated across many cells of a notebook, and long notebooks become time-consuming to navigate [20]. Important results accidentally get overwritten or deleted [29]. Data analysis is often fundamentally collaborative as analysts share notebooks, code snippets, documents, slides, images, and web pages [16, 20, 29]. Though when notebooks become too messy, analysts don't feel comfortable sharing their insights as is with others without substantial cleanup effort [27, 29].

To manage these messes, analysts have diverse strategies to clean their code. Many delete cells they no longer need, consolidate smaller cells into larger cells, and delete full analyses that did not turn out helpful. Long notebooks are abandoned for "fresh" ones with only a subset of successful parts from the long ones. Analysts organize code as they build it, some coding from top to bottom, some adding cells where they extend old analyses, some placing functions at the top, and some placing them at the bottom [20]. They add tables of contents, number sections, limit the size of cells, and split long notebooks into shorter ones [29].

Because of the challenges and tedium of managing messes, data analysts have given clear indications they need better tools to support the management of messes. In prior studies, analysts have asked for tools that let them collect scripts that can reproduce specific results; to compare outcomes from different versions of an analysis; and to recover a copy of a notebook that produces a version of a result [20]. They have also asked for tools that recover the history of how data was created, used, and modified [27]. This paper answers the call of these analysts, offering code gathering tools to help analysts manage messes in their notebooks.

Tools for cleaning messy code

Messiness is pervasive problem for code written in any language or domain. As such, researchers have designed tools to help coders—here, meaning software developers, end-user

programmers, analysts, and everyone else that writes code—rewrite unclear, bloated code. Recent tools have provided mixed-initiative interactions where coders work with their code editor to extract minimal, executable code examples that can reproduce a behavior [12] and large components of complex software projects [15]. Other tools let programmers clean application code by demonstrating by interacting with that application, and preserving only the executed code [23]. Such tools, like this work, leverage program slicing [34] to trace from code a coder indicated they want to preserve to other lines that should be included in a clean program. Code gathering tools uniquely aid with cleanup of notebook code by slicing a history of executed cell code, using the outputs of analysis code as a slicing criterion.

Another step of cleaning code is annotating it so others can understand, run, extend, and maintain it. Beyond text editing tools for writing comments and documentation, tools can help coders annotate code by helping them create walkthroughs visiting locations of interest in code projects [24, 32] and, in the context of notebooks, by helping coders fold and annotate cells with descriptive headers [27]. Tools such as these complement the code-centric cleaning utilities that code gathering tools currently provide.

Locating the causes of program output

Finding the code that is responsible for producing a result is complex when analysis produced many variants of code, generated long code files, or took place long ago. Amid these challenges, researchers have studied how tools can support analysts in locating the causes of program outputs. For example, tools can instrument programming languages (e.g., [25]), data analysis applications (e.g., [7]), and operating systems (e.g., [10]) to capture what results get produced, and the parameters, data, and code that produced them. Thereafter, this history is available and can be queried when analysts seek to recover the provenance of a result.

For programs more generally, prototype tools have been designed to help coders understand the causes of many types of program output, including graphics, console output, and bugs [21], interactive web page behavior [5, 6, 13, 14], user interfaces [4], and animations [9]. These tools help coders query for code by directly selecting outputs of interest [6, 9, 21], search for root causes of errors in code [21], record and replay ephemeral program behavior [5, 6], highlight executing lines of code [4], and filter code on a sliding scale of its relevance to a demonstrated behavior [14]. Code gathering tools similarly helps analysts find the code producing specific results from analysis, highlighting potentially disordered code dependencies in notebooks.

Managing and reviewing code versions

As analysts write code, they produce many versions of their code, from the resolution of snippets to files. These versions may be informally stored as comments [18] or notebook cells [20], leading to drawbacks like the difficulty of recalling promising versions of code, and code bloat. To help analysts manage versions, researchers have designed tools to make informal versioning of code easier, by enabling the versioning of small blocks of code directly within editors [18], automatically saving versions of cells upon each computation [26], and enabling exploratory programming on several simultaneous, linked versions of a code file [11].

These tools help analysts manage code versions amid fast-paced iterative programming. In dialogue with this work, code gathering tools support post-hoc mess management, by helping coders find, clean, and compare versions of code even if they spent no up-front effort on managing code versions or organizing their code. This approach is inspired by Janus' automatic versioning of code cells upon execution [26], and Variolite's archiving of runtime configurations [18]. Uniquely, code gathering tools enable analysts to recover and compare complete slices that produce results, even if they span many cells that appear out of order in a notebook.

In its intention to help analysts manipulate their programming history, code gathering tools builds on a tradition of research that shows that coders often consult programming histories to understand, debug, and extend their code [8] and rely on code's output, changelogs, and source code to find code to reuse [31]. Prior tools have supported in selectively undoing history [36], viewing the evolution of selected code snippets [30], and completing code with legacy names [22]. Code gathering tools seek to bring the mastery and understanding of history embodied by many of these prior tools into data analysts' computational notebooks.

3 A DEMO OF CODE GATHERING TOOLS

To convey the experience of using the code gathering tools in Jupyter Notebook, we describe a short scenario.[2] Consider an analyst, Dana, who is performing exploratory data analysis to understand variation and determiners of quality of a popular consumer good—chocolate. This section shows how code gathering tools could help her find, clean, and compare versions of code during data analysis.

Prologue: A proliferation of cells

Dana starts her analysis by loading a dataset, importing dependencies, and filtering and transforming the data. She writes code to display tables so she can preview the data, and displays charts and statistics that summarize the data's properties. To better understand key features of the data, she

[2] See also this paper's video figure.

builds a model to predict chocolate quality from the other features. Through experimentation, she tailors the model parameters to learn more about the features. Throughout the analysis, she makes messes, overwriting old code, deleting code that appears irrelevant, running cells out-of-order, and accumulating dozens of cells full of code and results. Dana starts to have trouble finding what she needs in the notebook.

Finding the code that produces a result

After several hours building and testing models, Dana is satisfied with a version of the model, but then recalls a peculiarity of the data: one of the columns is a percentage multiplied by 10 to avoid decimal points (e.g. 73.5% is represented with the integer 735). Although Dana wrote some code to check for triple-digit numbers, she cannot remember if she ran this code on the dataset before training her model. Hours have passed since she touched any code at the top of her notebook, where she loaded and cleaned the data.

Because she has installed code gathering tools, Dana sees all variable definitions highlighted in blue and all visual outputs outlined in blue. These marks highlight all results of analysis, including tables, charts, console output, console errors, definitions of models, loaded datasets, and transformed copies of those datasets. The marks indicate that Dana can click on any of the results to see the code in the notebook computed it, directly or indirectly.

Dana wants to know whether she cleaned her percentage data before training her classifier. She clicks on the results of classification—in this case, the predictions in the variable `predictions`. After clicking this variable, all lines that were used to compute the variable's value—including code for prediction, model training, data preparation, data loading, and importing dependencies—are highlighted in light purple (Figure 2). All other lines remain unhighlighted. With the relevant code highlighted, Dana scrolls through the sprawling notebook to browse the highlighted lines, skipping over long sections of irrelevant code and results. She finds the code that transforms the percentage data, namely, a cell defining the function `normalizeIt` and the cell after it. Because these lines are highlighted, Dana knows that she cleaned the percentage column before classification.

Removing old and distracting analysis code

Dana now has a notebook with a model that she likes—and much more code she no longer needs. Now that Dana knows what her data looks like and has a working set of data filtering, data transformation, and model training code, the code she wrote to visualize the data and debug her APIs will just get in the way. Dana decides to clean her notebook to a state where it only has the useful model-building code.

To clean the notebook, Dana clicks on a few results she wants to still be computed in the cleaned notebook, namely,

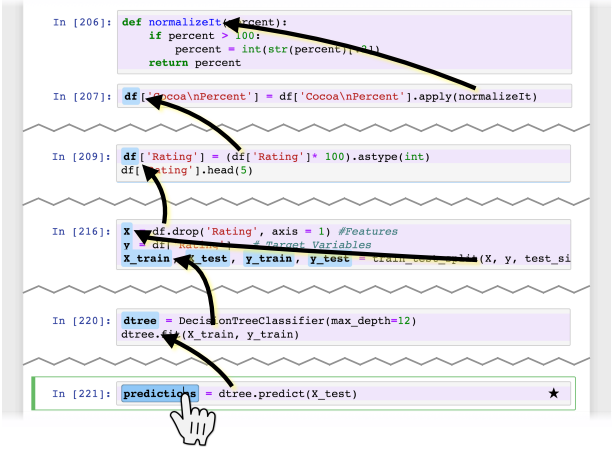

Figure 2: Finding relevant code with code gathering tools. With code gathering tools, an analyst can click on any result, and the notebook highlights in light purple just those lines that were used to compute the result or variable. The highlights appear throughout the notebook (which is condensed in the figure). Black arrows have been added in the figure to indicate the data dependencies that cause each line to be included in the highlighted set.

the classification results in the variable `predictions` and a bar chart showing the range of chocolate qualities used to build the classifier (Figure 3.2). Dana gets a sense of the size of the final cleaned notebook by looking at which lines in the notebook are highlighted as she selects each result. Then, Dana clicks the "Gather to Notebook" button (Figure 3.3), which opens a new notebook with the definition of `predictions`, the bar chart of chocolate quality, and the other code needed to produce these two results. The new, cleaned notebook has 16 cells, instead of the 47 in her original notebook. It contains the bar chart and omits 28 other visual results in the original. This reduces the overall size of the notebook from 13,044 to 1,248 vertical pixels in her browser, which is much easier to scroll through when editing the code (Figure 3.4). This cleaned notebook is guaranteed to replicate the results, as the tool reorders cells and resurrects deleted cells as necessary to produce the selected results. Dana verifies that running this notebook start-to-finish indeed replicates the chosen predictions and bar chart.

Reviewing versions of a result and the ordered, minimal code slices that produced them

To build a better predictor, Dana has been experimenting with different parameters to a decision tree classifier, like its maximum allowable depth and the minimum samples per branch. Dana remembers that she had previously created a simple, shallow decision tree with promising performance, but has not yet found a model with better performance.

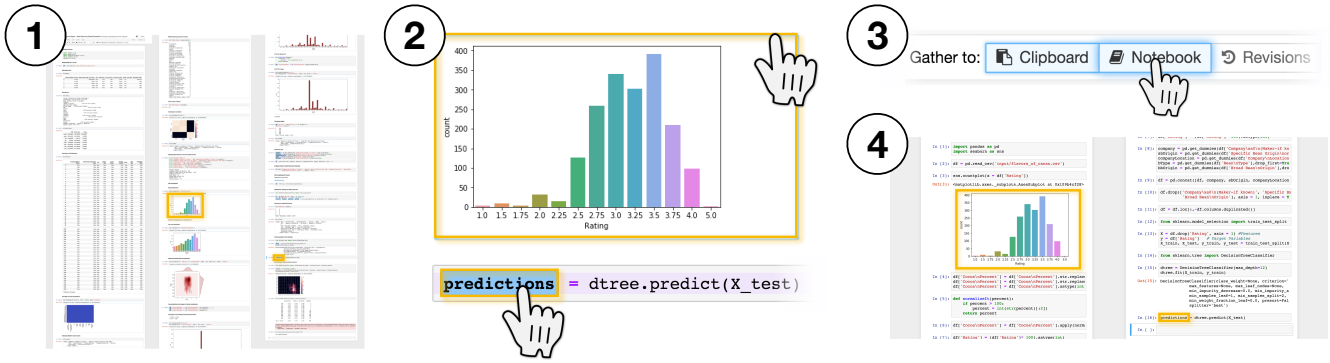

Figure 3: Cleaning a notebook with code gathering tools. Over the course of a long analysis, a notebook will become cluttered and inconsistent (1). With code gathering tools, an analyst can select results (e.g., charts, tables, variable definitions, and any other code output) (2) and click "Gather to Notebook" (3) to obtain a minimal, complete, ordered slice that replicate the selected results (4).

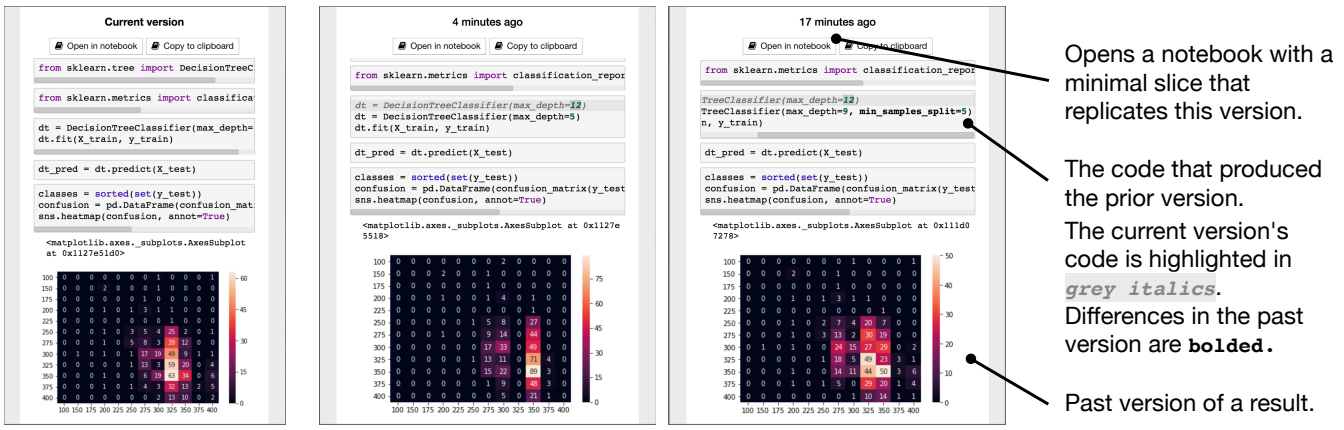

Figure 4: Comparing versions of a result with code gathering tools. When an analysts executes a cell multiple times, code gathering tools archive each version of the cell. When the analyst chooses the cell's output—say, the confusion matrix shown above—and clicks "Gather to Revisions," a version browser appears that lets them see all versions of that output, compare the code slices that produced each version, and load any of these slices to a new notebook, where the version's results can be replicated.

With code gathering tools, Dana can summon all past versions of her classifier's results and compare the code she used to produce these results. To do this, she clicks on a result—namely, a confusion matrix which visualizes the accuracy of the decision tree for each class—and then on the "Gather to Revisions" button. This brings up a version browser (Figure 4). Here, Dana sees all the versions of the result, arranged from left to right, starting with the current version and ending with the oldest version. Each version includes the relative time the result was computed, the code slice that produced that version of the result, and the result itself, in this case a confusion matrix.

Scrolling horizontally to access older versions, Dana finds several examples of decision trees with comparatively good accuracy. Among these alternatives, she searches for a shallow version of the decision tree with high accuracy. Dana looks above each result to see the code that produced it. The code is segmented into the cells that were originally executed, omitting lines that were not used to compute the result. Dana is familiar with the most recent version of the code and quickly finds the differences in past code versions by looking for bold code with a grey background. By seeing each version of the confusion matrix together with the model parameters that produced it, Dana finds the model she is looking for—a shallow tree with good performance, specifically with a depth of 3 and a minimum branch size of 5. The code that produced this version can be copied as cells or text to the clipboard, or opened as a new notebook that replicates that version; Dana opens a notebook with this version so she can refer back to it later.

Cleaning finished analysis code

Dana finished her data analysis and wants to share the results with an analyst on her team who can check her results and suggest improvements. However, the notebook is once again cluttered with code that would distract her colleague. While Dana wants to save her long and verbose notebook for her personal use later, she also wants a clean and succinct version of the notebook for her colleague. She chooses the prediction results of her model, clicks "Gather to Notebook," and saves the generated notebook to a folder shared with her colleague.

Exporting analysis code to a standalone script

After refining her analysis with her colleague, Dana wants to export a script that can be packaged with an article she is writing, so that others can replicate her results in their preferred Python environments. To do this, Dana selects the code that produces the results she wants her script to replicate, clicks "Gather to Clipboard," and then pastes the gathered code into a blank text file. This script replicates the results Dana produced in her notebook.

4 DESIGN MOTIVATIONS

The choice to build these specific code gathering tools was informed by formative interviews with eight data analysts and builders of tools for data analysis at a large, data-driven software company. During these interviews, we proposed several extensions to the notebook interaction model, and data scientists expressed the most enthusiasm for tools to help them clean their results, and explore past variants of their code. In this section, we highlight a few of the major design ideas motivating the current design of code gathering tools, which may be helpful for others designing tools for managing messes in computational notebooks:

Post-hoc management of messes. Given analysts' diverse personal preferences of whether and how to organize and manage versions of code, we decided code gathering tools should assist analysts regardless of such preferences. Whether they prefer to overwrite or save code versions explicitly, the tools still silently collect history, and provide access to the code that produced any visible result.

Portability of gathered code. Analysts reuse a notebook's code in that notebook, other notebooks, and scripts [20]. To support diverse code destinations, it is equally easy to gather code into cells, new notebooks, and lines of text. Analysts can "gather to notebook," which opens gathered cells in a new notebook, or they can "gather to clipboard," which simultaneously copies the cells to Jupyter Notebook's internal clipboard, and the code text to the system clipboard.

Query code via direct selection of analysis results. Prior research shows that programmers frequently look to program output when searching for code to reuse [31]. In notebooks, visual results break up walls of monospace text, providing beacons. We anticipated that selections of results would provide the most direct method for accessing relevant history.

5 IMPLEMENTATION

A computational notebook uses an underlying language interpreter (like Python). At any time, an analyst can submit any cell's code to the interpreter, in any order. The results that the interpreter produces and that the notebook displays depend on the order in which the analyst submits cell code. Hence, in the notebook context, a notebook's "program" is not the content of the notebook's cells, but the content of the cells that the analyst runs, in the order in which the analyst runs them. We call this the *execution log*.

We define code gathering as the application of program slicing to an execution log to collect ordered, minimal subsets of code that produced a given result. Program slicing is a static analysis technique wherein, given a target statement (called the slicing criterion), program slicing computes the subset of program statements (called the slice) that affect the value of the variables at the target statement [34]. In the notebook context, the variables/outputs that an analyst selects are the slicing criteria, and the gathered code is the slice. We implemented code gathering as a Jupyter Notebook extension with roughly 5,000 lines of TypeScript code. Our implementation supports notebooks written in Python 3. The details in this section could serve as a conceptual template for tool builders seeking to support code gathering for notebooks in other Pythonic languages like Julia and R.

Collecting and slicing an execution log

To find the code that produces a result, the tools first need a complete and ordered record of the code executed in the notebook. We build such a record, the "execution log," by saving a summary of each cell as it is executed. A cell summary contains two parts: first, the cell's code, which will be joined with the code of other cells into a temporary program used to find code dependencies; second, the cell's results, which can be used as slicing criteria, and shown in a version browser as the output of running that cell.

The code for some cells, if included in the execution log, will cause errors during program slicing. Namely, if the code that contains syntax errors, the temporary program used during dependency analysis will fail to parse; if it raises run-time errors, a slice containing that cell might raise the same error. Therefore, cells with syntax errors and runtime errors are omitted from the log. Ignoring cells with parse errors is consistent with Jupyter's semantics: if an executed cell contains any parse errors, all of its code is ignored by the interpreter. Ignoring cells with run-time errors is inconsistent with Jupyter's semantics, in that the interpreter will run

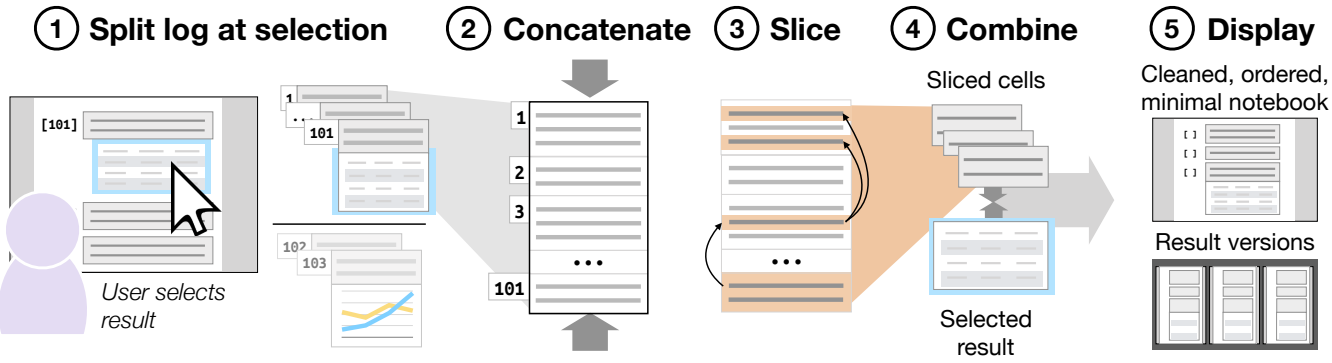

Figure 5: Implementation of code gathering. When an analyst wants to gather the code that produced a result, the code gathering backend splits the log of executed cells at the last cell where the analyst clicked a result, and discards the other cells (1), concatenates the text from the remaining cells into a program (2), slices the program using the analyst's selections as a slicing criterion (3), combines the sliced cells with the selected results if they are code outputs (4), and displays these cells in a notebook or a version browser (5).

the statements up to the point where the error occurs. This limitation does not cause problems in practice, since analysts typically correct such errors and re-run the cells.

Next, we slice the execution log to provide code slices that replicate results. When an analyst selects results in a notebook, they specify slicing criteria. When they select a variable definition, they add the statement containing the variable definition as a slicing criterion. When they select a cell's output, they add all statements in that cell.

To slice the execution log, there must first be a "program" to slice. We build such a program by filtering the log to the cells were executed before the cells containing slicing criteria: these cells won't be included in the slice, and would unnecessarily slow down the slicing algorithm. Then, the program is built by joining the text of the remaining cells, in the order they were executed (Figure 5.1–2).

Finally, we slice the program (Figure 5.3). We implemented a standard program slicing workflow—parsing the program with a Jison-generated parser [1]; searching the parse tree for variable uses, definitions, and control blocks; computing control dependencies (e.g., dependencies from statements to surrounding if-conditions and for-loops) and data dependencies (e.g., dependencies from statements using a variable to statements that define or modify that variable); and slicing by tracing back from the slicing criteria to all the statements they depend on. When computing data dependencies, we determine if methods modify their arguments by looking up this information in a custom, extensible configuration file containing data dependencies for functions from common data analysis libraries (e.g., pandas, matplotlib).

Our current implementation supports interactive computation times by splitting slicing into small, reusable parts: when a cell is executed, its code is immediately parsed, and its variable definitions and uses detected. With these precomputed pieces of analysis, gathering takes place at interactive speeds, as the most costly analyses have been performed before the analyst gathers any code.

6 IN-LAB USABILITY STUDY

We designed a two-hour, in-lab usability study to understand the support that code gathering tools can provide data analysts as they write code in computational notebooks. We were fairly confident of the ability of code gathering tools to eliminate clerical work—like the removal of irrelevant code, or recovery of dead code—given the design of the tool and evidence from several prior pilot studies. Therefore, the questions we sought to answer focused on the match between the control analysts desired over messy notebooks, and the support code gathering tools currently provide. We therefore designed our study to answer these research questions:

RQ1. What does it mean to "clean"? When we ask analysts to clean a notebook, what do they do? Could code gathering tools support the work they are doing?

RQ2. How do analysts use code gathering tools during exploratory data analysis? In our design of the tools, we hypothesized that analysts would use the tools for highlighting code, gathering to notebooks, and version browsing to find, clean, and compare versions of code. Do they?

We invited 200 randomly selected data analysts at a large, data-driven software company. The invitation stated the requirement of experience with Jupyter notebooks and Python. We recruited 12 participants altogether (3 female, aged 25–40 years, median age = 29.5 years). Participants reported the following median years of experience on a Likert scale: 6–10 years programming; 3–5 years programming in Python; and

1–2 years using Jupyter Notebooks. Five participants report using Jupyter Notebooks daily; three, weekly; one, monthly; and three, less than monthly. We compensated participants with a US$50 Amazon gift card.

Tasks. To start, each participant signed a consent form and filled out a background questionnaire. The session then consisted of two cleaning tasks and a exploratory data analysis task. For the two cleaning tasks, we gave participants two existing notebooks from the UCSD Jupyter Notebook archive [28], one about Titanic passengers, one about the World Happiness Index. We chose these notebooks because they are in Python, execute without errors, use popular analysis and visualization libraries, involve non-technical domains, and are long enough to be worthy of cleaning. We counterbalanced use of the two notebooks between subjects.

For the first cleaning task, we asked the participant to scan the notebook for an interesting result and to clean the notebook with the goal of sharing that result with a colleague (10 minutes). After a brief tutorial about code gathering, we then asked the participant to repeat the cleaning task on a different notebook, this time using the code gathering features (10 minutes). Finally, for the exploratory task, we gave participants a dataset about Hollywood movies and asked them to create their own movie rankings, ready for sharing (up to 30 minutes). We chose this dataset as we thought it would understandable and interesting to analysts from a wide variety of backgrounds. During all tasks, participants could use a web browser to search the web for programming reference material. After each of the three tasks, the participant filled out a questionnaire: the first about how they currently clean notebooks; the second about the usefulness of code gathering tools for notebook cleaning; and the third about the usefulness of code gathering tools for data exploration. Throughout the tasks, we encouraged participants to think aloud, and we transcribed their remarks.

Each participant used an eight-core, 64-bit PC with 32 GB of RAM, running Windows 10, with two side-by-side monitors with 1920 × 1200 pixels. One monitor displayed Jupyter Notebooks; the other displayed our tutorial and a browser opened to a search engine.

7 RESULTS

In the section below, we refer to the 12 analysts from the study with the pseudonyms P1–12.

The meaning of "cleaning"

Before giving analysts the tutorial about code gathering tools, we first asked them to describe their cleaning practice and to clean a notebook in their usual way. This allowed us to understand their own interpretation of "cleaning" before biasing them with our tool's capabilities. Many analysts explained "cleaning" in a way that is compatible with code gathering, namely keeping a desired subset of results while discarding the rest (P8, P10–12). Indeed, one analysts's description of cleaning is surprisingly close to the code gathering algorithm: "So I picked a plot that looked interesting and that's maybe something I would want to share with someone and then, if you think of a dependency tree of cells, sort of walked backwards, removed everything that wasn't necessary" (P10).

In their everyday work, some analysts clean by deleting unwanted cells, but most copy/paste desired cells to a fresh notebook. (One analysts who cleans by deletion initially found the non-destructive nature of code gathering to be unintuitive, but adjusted after practice (P4).) Many described the process as error-prone and frequently re-execute the cleaned notebook to check that nothing is broken.

Every analyst reported that choosing a subset of cells is part of the cleaning process. However, for several analysts, "cleaning" includes additional activities. Several analysts reported that cleaning involves a shift in audience from oneself to other stakeholders, like peers and managers (P1, P5–7, P11). Hence, cleaning involves adding documentation (comments or markdown) (P1, P5, P7, P10, P11) and polishing visualizations (e.g., adding titles and legends) (P1, P6). Some analysts reported that cleanup includes improving both notebook quality (e.g., merging related cells (11) and eliminating unwanted outputs (P3, P6)) and code quality (e.g., eliminating (P3, P6) or refactoring (P3, P4, P12) repeated code). Finally, for some, cleaning involves integrating the code into a team engineering process, for example, by checking the code into a repository or turning it into a reusable script (P7).

Code comments. When code is gathered, code comments and markdown are removed, as they are unnecessary to replicating a result. One analyst believed this made the code look cleaner (P5). Yet at least two others believed that comments, when close to the code, should be included by default (P3, P10). One of these analysts noted that including irrelevant comments would not be problematic, as "it's easy to remove some extraneous text" (P10).

How analysts use code gathering tools to support exploratory data analysis

After both the second notebook cleaning task and the exploratory analysis task, we asked analysts to provide subjective assessments of code gathering, broken down into seven features (Figure 6). Gathering code to a new notebook is the clear favorite, with nearly every analyst rating it as "very useful" for both tasks. The dependency highlights were also popular. Many analysts did not find opportunities to try the version browser during the two tasks, likely due to the short duration of the lab session. Similarly, many analysts did not experience the recovery of deleted code, either because

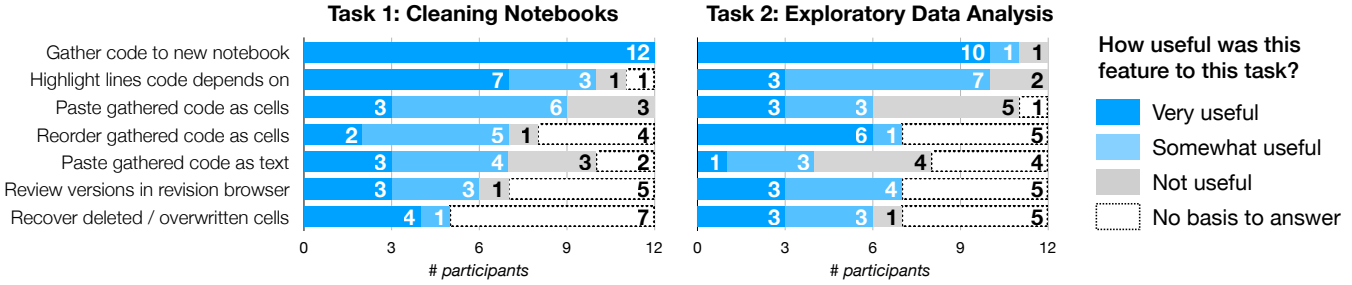

Figure 6: Analysts found code gathering tools most useful for gathering code to new notebooks, when they cleaned notebooks, and when they performed exploratory analysis. Analysts also appreciated dependency highlights, especially when they were cleaning code.

no relevant code was deleted or because the user interface recovers deleted code silently.

Valued and versatile feature of gathering code to new notebooks. Nine analysts gathered code to a new notebook at least once during the exploratory task. Analysts gathered code to a notebok a median of 1.5 times (σ = 3.7) during this task, with one analyst even gathering notebooks 12 times (P3). Analysts most often gathered code to a notebook for its intended purpose of cleaning up their code as a "finishing move" after exploration (P6). Analysts clearly valued this aspect of the tool, calling it "amazing" and "beautiful" (P10), that they "loved it" (P5), it "hits the nail on the head" (P9), and will save them "a lot of time" (P11).

Analysts saw additional value in gathering code to notebooks beyond our original design intentions. During the exploratory task, one analyst used gathering to a new notebook as a lightweight branching scheme. As he explored alternatives, he would gather his preferred alternative to a new notebook to create a clean slate for further exploration (P3). Another analyst used gathering as a way to generate reference material. She created data visualizations, then gathered them to new notebooks, so she could quickly flip to the visualizations as she carried on exploring in her original notebook (P4). Finally, one analyst used gathering to support cleaning for multiple audiences. At the end of the exploratory task, he gathered many visualizations to one notebook and documented them for his peer data analysts; he then gathered his movie ranking result to a different notebook intended for those who only want to know the final answer (P2).

Analysts were eager to incorporate gathering into their data analysis workflows: seven of twelve analysts asked us when we would release the tool. One analyst envisioned gathering becoming part of code-cleaning parlance: "once this is public, people will send you bloated notebooks. I'll say, nope, you should gather that" (P10).

Use of the dependency highlighting. During the exploratory task, 8 analysts clicked on at least one variable definition, and 9 clicked on at least one output area. Additionally, during the cleaning tasks, as these tasks involved reading unfamiliar notebooks, a few analysts used the dependency highlights as a way to understand the unfamiliar code.

Use and disuse of the version browser. Two analysts opened the version browser at least once (P2, P3). Both copied the cells to the clipboard from a version in the version browser at least once; one analyst in fact copied cells for versions four times during their session (P2). The other analyst opened a version in a new notebook. This analyst wanted to compare versions of a cell that sorted data based on two different dimensions, and used the version browser to recover code from a prior version without overwriting the current cells, which they wished to preserve (P3).

Some analysts who did not use the version browser believed that they might eventually use it in their own work (P6, P8, P9). One analyst noted Jupyter Notebook's implementation of "undo" is not sufficient for them, and the version browser could provide some of the backtracking functionality they want (P6). Another reported that the version browser could be useful in their current work, where they have iteratively developed an algorithm and are managing three notebooks containing different versions of analyses (P9). However, two analysts believed they wouldn't use the version browser, as its view of versions is too restrictive. The version browser collects versions ending with multiple executions of the same cell, yet these analysts preferred to modify and re-run old analyses in new cells (P10, P11).

Downsides and gaps. A few analysts mentioned that repeatedly gathering code to a new notebook creates a different kind of mess, namely clutter across notebooks, rather than clutter within a notebook. For example, gathering multiple times typically causes initialization code (e.g. loading the dataset) to be duplicated in each generated notebook (P3, P4, P6). In effect, a notebook and the notebooks gathered from it form a parent/child relationship that the user interface does not currently recognize. Analysts suggested several improvements. First, gathering to a new notebook should create a provisional notebook, rather than being saved by default,

and its name should be related to the original notebook's name. One analyst suggested linked editing across this family of notebooks as a way to deal with duplicated code. For example, renaming a variable in one family member could automatically rename it in all members (P12).

Validating the design motivations

Analysts' feedback offered evidence of the role that our design motivations played in the usefulness of the tools:

Post-hoc management of messes. Analysts valued the ability to manage messes without up-front effort to organize and version code. This was a benefit of gathering to new notebooks, as analysts appreciated simple affordances to clean up their messy analysis code (P1, P2, P6, P8, P9). For one analyst, the tool encouraged them, for better or worse, to "not to care… too much about data cleaning or structure at this moment. I say it was nice in a way, that I can just kind of go on with what I want to do" (P12). For some analysts, this was the downside of the version browser, which required them to run new versions of code in the same cell (P10, P11).

Portability of gathered code. Analysts reused gathered code by opening fresh notebooks, pasting cells, and pasting plaintext. In the exploratory task, nine analysts gathered code to notebooks, and five gathered code to the clipboard, to paste as either cells or plaintext. By pasting plaintext into one cell, analysis code looked "a lot cleaner" (P3), and several analysts wanted an easier ways to gather code to scripts. Others preferred pasting code as distinct cells (P5, P7). One analyst simply liked having the choice (P10).

Querying code via direct selection of analysis results. Analysts appreciated the directness with which they could gather code: "It was very easy to just click, click, click on something and then grab the code that produced" a result (P10). The directness allowed analysts to clean their code by asking, "what do I need?" rather than "what do I not need?" (P3).

Limitations

Our study has two limits to external validity, which are common in lab-based usability evaluations: first, the participants did not do their own work, on their own data, in their own time frame. We created realistic tasks by choosing notebooks and datasets from the UCSD Notebook Archive, itself mined from GitHub. Ideally, participants would use their own data and analyses. However, several informants in our formative interviews said their data was too sensitive for us to observe, so we did not pursue this option. The second limitation is the study's short duration, which we believe accounts for the low use of the version browser feature. As P6 commented, "the other features will be more valued for notebooks that have been used for a long time/long project."

8 DISCUSSION AND FUTURE WORK

There are several simplifying assumptions embodied in the prototype, which could be made more flexible in future tools.

Flexible versions of results. The Revisions button shows different versions of a result over time, but this raises a question: when should we consider two results in the execution log to be different versions of the "same" result? Our prototype uses the identity of the cell that produces a result to determine "sameness". This heuristic works for many styles of exploratory programming, but lacks flexibility. Future tools could determine "sameness" with heuristics based on text content, rather than notebook structure.

Long-lived execution histories. The code gathering tools' execution log lasts only for a single programming session, which limits the scope of the Revisions button and resurrecting code from deleted or overwritten cells. Future tools could create a persistent execution log, both for individual users and for sharing. Further, transferring code or cells between notebooks (e.g., copy/paste) could also transfer the relevant execution steps. This creates both technical challenges (e.g. merging log segments into a coherent history) and user interface challenges (e.g. scaling the Revisions display to months' or years' worth of revisions).

Cleaning incrementally. Given the fluid nature of notebook programming, we designed the use of code gathering tools to be entirely *post hoc*, i.e. a user never needs to anticipate that cleaning will happen in the future. While gathering meets this goal, the gathering step requires the user to select all the results to be saved at once. Several participants wanted to create a cleaned notebook in a series of steps, that is, the ability for a gathering step to "patch" notebooks gathered in a previous step. Future tools could use algorithms from revision control systems to support this flexibility.

Conclusions

Our qualitative usability study with 12 professional data scientists confirmed that cleaning computational notebooks is primarily about removing unwanted analysis code and results. The cleaning task can also involve secondary steps like improving code quality, writing documentation, polishing results for a new audience, or creating scripts. Participants find the primary cleaning task to be clerical and error-prone. They therefore responded positively to the code gathering tools, which automatically produces the minimal code necessary to replicate a chosen set of analysis results, using a novel application of program slicing. Analysts primarily used code gathering as a "finishing move" to share work, but also found unanticipated uses like generating reference material, creating lightweight branches in their code, and creating summaries for multiple audiences.

REFERENCES

[1] Jison. https://jison.org Accessed September 8, 2018.

[2] Jupyter. http://jupyter.org/ Accessed September 2, 2018.

[3] Unofficial Jupyter Notebook Extensions. https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/. Accessed September 3, 2018.

[4] Joel Brandt, Vignan Pattamatta, William Choi, Ben Hsieh, and Scott R. Klemmer. 2010. Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code. (2010). Stanford technical report.

[5] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 473–484.

[6] Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 259–268.

[7] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: Visualization meets Data Management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 745–747.

[8] 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.

[9] Paul A. Gross, Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher. 2010. A Code Reuse Interface for Non-Programmer Middle School Students. In *Proceedings of the 15th international conference on Intelligent user interfaces*. ACM, 219–228.

[10] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance (TaPP'12)*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=2342875.2342882

[11] 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*. ACM, 91–100.

[12] Andrew Head, Elena L Glassman, Björn Hartmann, and Marti A Hearst. 2018. Interactive Extraction of Examples from Existing Code. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM.

[13] Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 270–279.

[14] Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-Tuned Discovery of Interactive Web UI Feature Implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 233–245.

[15] Reid Holmes and Robert J. Walker. 2012. Systematizing Pragmatic Software Reuse. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 4 (2012), 20.

[16] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization & Computer Graphics* 12 (2012), 2917–2926.

[17] Kyle Kelley and Brian Granger. 2017. Jupyter frontends: From the classic Jupyter Notebook to JupyterLab, nteract, and beyond. (2017). JupyterCon. A video of this talk is available at: https://www.youtube.com/watch?v=YKmJvHjTGAM.

[18] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 1265–1276.

[19] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2017 IEEE Symposium on*. IEEE, 25–29.

[20] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM.

[21] Andrew Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1569–1578.

[22] Yun Young Lee, Darko Marinov, and Ralph E. Johnson. 2015. Tempura: Temporal dimension for ides. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 212–222.

[23] Josip Maras, Maja Štula, Jan Carlson, and Ivica Crnković. 2013. Identifying Code of Individual Features in Client-Side Web Applications. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1680–1697.

[24] Christopher Oezbek and Lutz Prechelt. 2007. JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 64–73.

[25] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2016. Fine-Grained Provenance Collection over Scripts Through Program Slicing. In *International Provenance and Annotation Workshop*. Springer, 199–203.

[26] Adam Rule. 2018. *Design and Use of Computational Notebooks*. Ph.D. Dissertation. UC San Diego.

[27] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. (2018).

[28] Adam Rule, Aurélien Tabard, and James D. Hollan. Data from: Exploration and Explanation in Computational Notebooks. https://doi.org/10.6075/J0JW8C39.

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

[30] Francisco Servant and James A. Jones. 2012. History Slicing: Assisting Code-Evolution Tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM.

[31] 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*. ACM, 3509–3521.

[32] Ryo Suzuki. 2015. Interactive and Collaborative Source Code Annotation. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 799–800.

[33] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

[34] Mark D. Weiser. 1979. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. Ph.D. Dissertation. https://search-proquest-com.libproxy.berkeley.edu/docview/302949655?accountid=14496 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-06-08.

[35] YoungSeok Yoon and Brad A. Myers. 2012. An Exploratory Study of Backtracking Strategies Used by Developers. In *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*. IEEE Press, 138–144.

[36] YoungSeok Yoon and Brad A. Myers. 2015. Supporting Selective Undo in a Code Editor. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 223–233.