Managing Messes in Computational Notebooks

Andrew Head
UC Berkeley
andrewhead@berkeley.edu

Fred Hohman
Georgia Institute of Technology
fredhohman@gatech.edu

Titus Barik
Microsoft
titus.barik@microsoft.com

Steven M. Drucker
Microsoft Research
sdrucker@microsoft.com

Robert DeLine
Microsoft Research
rob.deline@microsoft.com

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

KEYWORDS

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

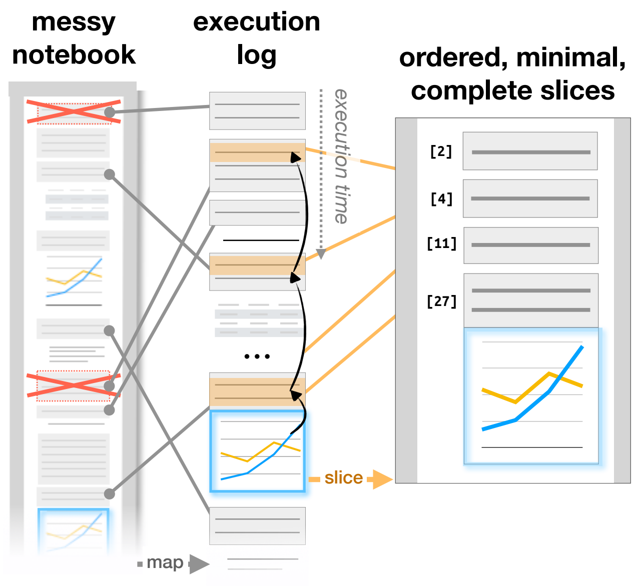


Figure 1: Code gathering tools help analysts manage programming messes in computational notebooks. The tools map selected results (e.g., outputs, charts, tables) in a notebook to the ordered, minimal subsets or "slices" of code that produced them. With these slices, the tools help analysts clean their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

ACM Reference Format:
Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *CHI Conference on Human Factors in Computing Systems Proceedings (CHI 2019), May 4–9, 2019, Glasgow, Scotland UK*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3290605.3300500

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" [21, 33], containing "ugly code" and "dirty tricks" in need of "cleaning" and "polishing" [33].

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. For the millions of users of notebooks [17], such messes are quite common: for instance, nearly half of public notebooks on GitHub include cells that were executed in a different order than they are listed [30]. Messes make it difficult for an analyst to navigate and understand their code, and to recall how results (e.g., charts, tables) were produced. Messes also make analysts reluctant to share their analyses with stakeholders and collaborators [21, 33].

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 technique called program slicing [38], 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 (Figure 1).

This paper makes two contributions. The first contribution is the design and implementation of code gathering tools. 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 [15], a popular notebook with millions of users [17]. The extension is available for use as a design artifact and as a practical tool for exploratory data analysis in notebooks.

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 [37], folding and unfolding cells [31], and marking and tagging lightweight versions of snippets [18]. 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 messes in notebooks. 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 lightweight branching. We also refined our understanding of the meaning of "cleaning," and how code gathering tools support an important yet still incomplete set of tasks analysts consider to be 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 means to manage these messes.[1]

2 BACKGROUND AND RELATED WORK

Messes in computational notebooks

Lackluster code quality appears to be intrinsic to exploratory programming. Analysts regularly prioritize the efficient discovery of solutions over writing high-quality code [19]. They clutter their programs by saving old versions of their code in comments [18, 39]. 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 output from their code, and debug their code; and then combine and eliminate cells as their analyses reach completion [21].

Eventually, messes get in the way of data analysis. It becomes difficult to understand analyses split across many cells of a notebook, and long notebooks become time-consuming to navigate [21]. Important results accidentally get overwritten or deleted [33]. While analysts often wish to share their findings with others [16, 21, 33], they are often reluctant to do so until they have cleaned their code [31, 33].

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

[1]See the project web page, https://microsoft.github.io/gather/, for installation instructions for the extension, and study materials.

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 [21]. They add tables of contents, assign numbers to sections, limit the size of cells, and split long notebooks into shorter ones [33].

Because of the challenges and tedium of managing messes, data analysts have clearly indicated 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, compare outcomes from different versions of an analysis, recover copies of notebooks that produce a version of a result [21], and to recall the history of how data was created, used, and modified [31].

Tools for cleaning messy code

Messiness is pervasive problem in code written with any language or tool. As such, researchers have designed tools to help programmers clean bloated code. Recent mixed-initiative tools have been designed to help programmers extract minimal, executable code examples [9] and self-contained software components [13, 25] from existing code projects. Such tools, like ours, use program slicing [38] to find code that should be preserved and removed as code is cleaned. Code gathering tools help analysts clean notebook code by slicing a history of executed cell code. *Verdant*, like our tools, helps analysts collect “recipes” of notebook code that reproduce selected results using program slicing [20]. Uniquely, our paper describes affordances for cleaning, navigating, and browsing code versions in notebooks using such slices.

Automated refactoring tools are a common type of code cleaning tool available in popular programming IDEs [26]. Refactoring tools help programmers improve the readability and maintainability of their code by assisting with behavior-preserving code transformations, like renaming variables and extracting code into functions. Code gathering tools provide a type of refactoring for notebooks, letting analysts rearrange code so it reproduces selected outputs. We draw inspiration from direct interactions in recent refactoring tools developed by human-computer interaction researchers [10, 23].

Another task in cleaning code is annotating it so others can understand, run, extend, and maintain it. Tools can help programmers create walkthroughs of points of interest in code projects [27, 36] and, in the context of notebooks, help programmers fold cells and annotate them with descriptive headers [31]. Tools such as these complement the code-centric cleaning utilities that code gathering tools provide.

Locating the causes of program output

As their code bases get larger and analysts throw away old code, analysts can find it difficult if not impossible to recall the code they used to produce important results. To help analysts recover the provenance of important results, researchers have proposed tools that capture histories of an analyst’s activity, from their programming languages [29], exploratory data analysis applications [4], and operating system logs [7]. These tools then let programmers recall a result’s provenance by reviewing the captured logs.

The software development tools community has designed a number of tools to help programmers find code that produces many types of observed output. These tools highlight lines of code as a program executes them [1, 2, 28], or filters the code for a program to only the lines that were executed in a recent execution [3, 6, 11, 12]. They also help programmers trace through code backward from static outputs like console output and error logs [22]. Like these prior systems, code gathering tools help programmers locate code of interest, in this case within computational notebooks.

Managing and reviewing code versions

Programmers often consult code histories to understand, debug, and extend their code [5]. They search for code to reuse by inspecting prior versions’ outputs, changelogs, and source code [35]. Prior tools have helped programmers make use of history to selectively undo code changes [40], review the evolution of selected snippets [34], and auto-complete code using legacy names [24]. To help data analysts manage code versions, researchers have designed tools to help them save and navigate versions of code fragments [8, 18, 20], notebook cells [20, 30], and entire notebooks [20].

In dialogue with this work, code gathering tools support post-hoc mess management, by helping analysts 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 was inspired by Janus’ automatic versioning of code cells upon execution [30], and Variolite’s archiving of runtime configurations [18]. Our approach to versioning code differs slightly from prior tools. With Janus, versions are per cell, whereas versions in our tools are per “slice”; with Variolite, versions of code snippets are created with forethought, but in hindsight with code gathering tools.

3 DESIGN MOTIVATIONS

We conducted formative interviews with eight data analysts and builders of tools for data analysis at a large, data-driven software company. During the interviews, we proposed several extensions to the notebook interaction model. Analysts expressed the most enthusiasm for tools to help them clean their results, and explore past variants of their code. These conversations and a review of the related literature yielded several key ideas that guided our design of notebook cleaning tools. We refer to the analysts as D1–8 below.

Post-hoc management of messes. Analysts have diverse personal preferences of whether and how to organize and manage versions of code. The analysts we spoke to each had their own workarounds, like keeping cells ordered so they always reproduce the visible results (D7, D8), copying useful snippets to external files (D4), and assigning dataset variables new names every time they transform them to avoid overwriting the original data (D6). Some code organization strategies conflict with others: some analysts clean their notebooks as they write it, while others preserve a record of everything they have tried [21]—though you cannot do both in current notebooks. One analyst noted that you don't always know if you are creating versions of code until you already have (D7). We decided code gathering tools should assist analysts regardless of whether they think to organize their code, and whether they prefer to overwrite or save copies of old code. The tools 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 [21]. The analysts we spoke to wanted tools to help them reuse code in new notebooks (D7), to apply old notebooks' analyses to new data (D8), and to export code to other files (D4, D5). We designed our tools to make it equally easy to gather code to new notebooks, cells, and lines of text.

Query code via direct selection of analysis results. Prior research shows that programmers frequently look to program output when searching for code to reuse [35]. 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.

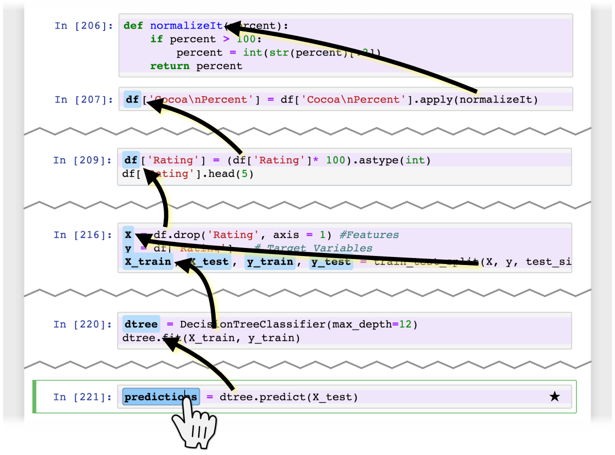
4 A DEMO OF CODE GATHERING TOOLS

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. To better understand key features of the data, she 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.

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



```
In [206]: def normalizeIt(percent):
              if percent > 100:
                  percent = int(str(percent)[:2])
              return percent

In [207]: df['Cocoa\nPercent'] = df['Cocoa\nPercent'].apply(normalizeIt)

In [209]: df['Rating'] = (df['Rating']* 100).astype(int)
          df['Rating'].head(5)

In [216]: X = df.drop('Rating', axis = 1) #Features
          y = df['Rating']    # Target Variables
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_si

In [220]: dtree = DecisionTreeClassifier(max_depth=12)
          dtree.fit(X_train, y_train)

In [221]: predictions = dtree.predict(X_test)
```

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.

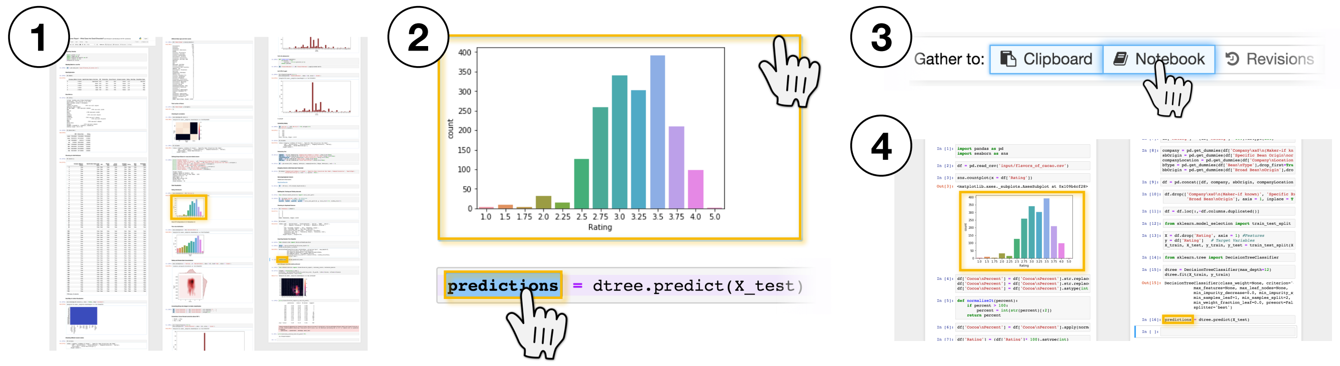
Finding the code that produces a result

After several hours building and testing models, Dana is satisfied with a version of the model, but then realizes there may be a problem with the model. One of the numeric fields contains erroneous values. Although Dana wrote code to fix these values, she cannot remember if she ran this code on the dataset that was used to trained the model.

Because she has installed code gathering tools, Dana sees all variable definitions (data frames, models, etc.) highlighted in blue and all visual outputs (console output, tables, figures, etc.) outlined in blue. She clicks on the results of classification (a variable named `predictions`) and then all lines that were used to compute the variable's value are highlighted in light purple (Figure 2). 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 column of unclean values 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 (Figure 3.1). Now that



```
predictions = dtree.predict(X_test)
```

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 replicates the selected results (4).

Dana knows what her data looks like and has a working set of data filtering, data transformation, and model training code, the code to visualize the data and debug the 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, the classification results in the `predictions` variable and a histogram 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.

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 and the result itself.

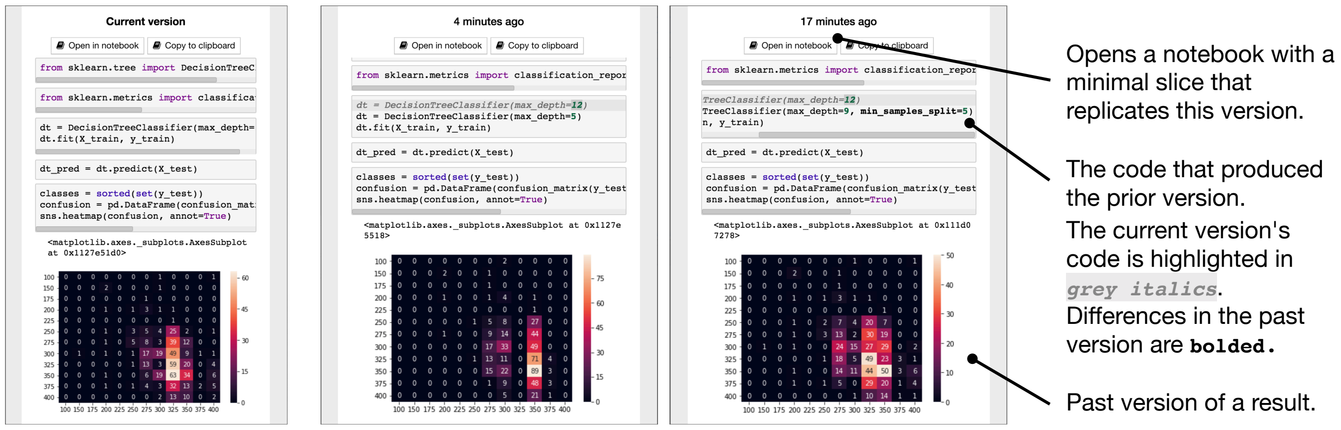
Scrolling horizontally to access older versions, Dana finds several examples of decision trees with comparatively good accuracy. Differences from the current version of the code are shown with bold text and a colored background. Dana finds the model she is looking for—a shallow tree with good performance. 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


Current version
Open in notebook
Copy to clipboard
4 minutes ago
Open in notebook
Copy to clipboard
17 minutes ago
Open in notebook
Copy to clipboard
Opens a notebook with a minimal slice that replicates this version.
The code that produced the prior version.
The current version's code is highlighted in grey italics. Differences in the past version are bolded.
Past version of a result.

```
from sklearn.tree import DecisionTreeC

from sklearn.metrics import classifica

dt = DecisionTreeClassifier(max_depth=
dt.fit(X_train, y_train)

dt_pred = dt.predict(X_test)

classes = sorted(set(y_test))
confusion = pd.DataFrame(confusion_mat
sns.heatmap(confusion, annot=True)

<matplotlib.axes._subplots.AxesSubplot
at 0x1127e51d0>
```

```
from sklearn.metrics import classification_repor

dt = DecisionTreeClassifier(max_depth=12)
dt = DecisionTreeClassifier(max_depth=5)
dt.fit(X_train, y_train)

dt_pred = dt.predict(X_test)

classes = sorted(set(y_test))
confusion = pd.DataFrame(confusion_matrix(y_test
sns.heatmap(confusion, annot=True)

<matplotlib.axes._subplots.AxesSubplot at 0x1127e
5518>
```

```
from sklearn.metrics import classification_repor

TreeClassifier(max_depth=12)
TreeClassifier(max_depth=9, min_samples_split=5)
n, y_train)

dt_pred = dt.predict(X_test)

classes = sorted(set(y_test))
confusion = pd.DataFrame(confusion_matrix(y_test
sns.heatmap(confusion, annot=True)

<matplotlib.axes._subplots.AxesSubplot at 0x111d0
7278>
```

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 into a new notebook, where the version's results can be replicated.

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.

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 [38]. 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 Python-like 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 contains syntax errors, the temporary program used during dependency analysis will fail to parse; if it raises runtime 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 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 produce 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 from 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

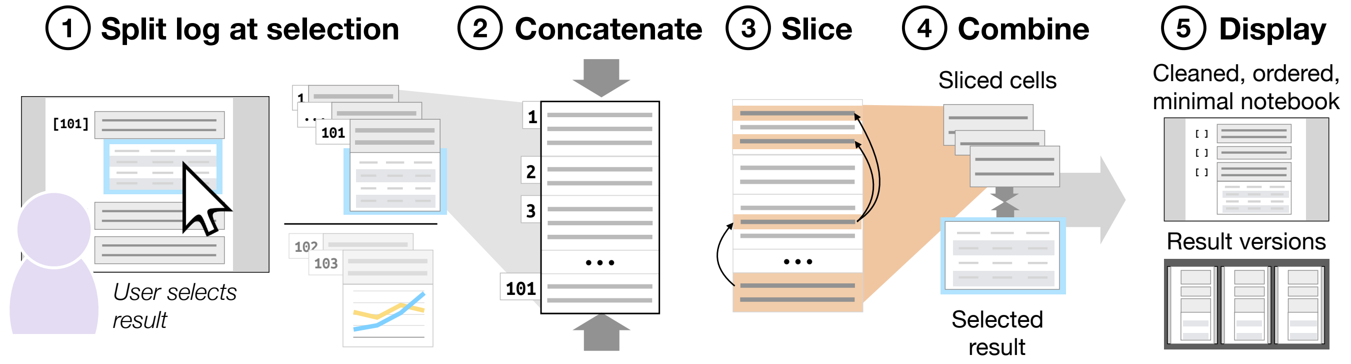


Figure 5: Implementation of code gathering. When an analyst wants to gather the code that produced a result, the code gathering backend splits the log of executed cells at the last cell where the analyst clicked a result, and discards the other cells (1), concatenates the text from the remaining cells 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).

exclude the cells that were executed after 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). This program may include the code of a single cell more than once, e.g., if the cell was executed twice to compute the chosen result.

Finally, we slice the program (Figure 5.3). We implemented a standard program slicing workflow—parsing the program with a Jison-generated parser [14]; 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 to 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 (aged 25–40 years, median age = 29.5 years, 3 female). Participants reported the following median years of experience on an ordinal scale: 6–10 years programming; 3–5 years programming in Python; and 1–2 years using Jupyter Notebooks. Five participants reported 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 an exploratory data analysis task. For the two cleaning tasks, we gave participants two existing notebooks from the UCSD Jupyter Notebook archive [32], one about Titanic passengers, and 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 be 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.

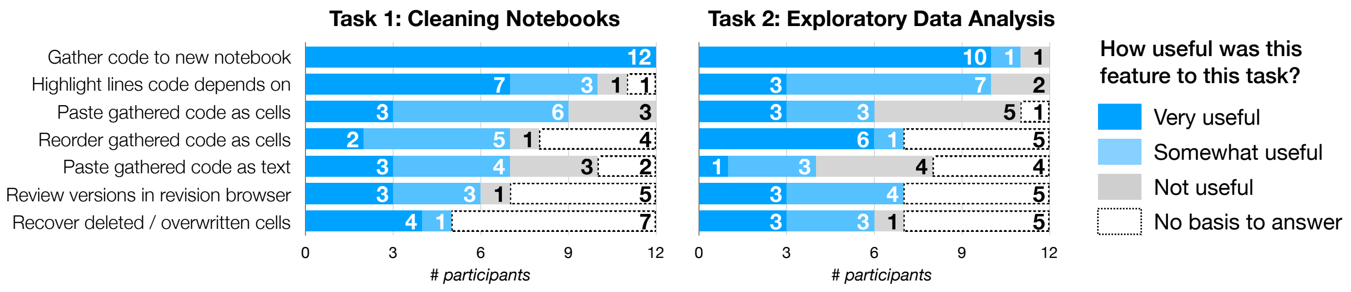
Every analyst reported that choosing a subset of cells is part of the cleaning process. However, for several analysts, "cleaning" includes additional activities. Several analysts reported that cleaning involves a shift in audience from oneself to other stakeholders, like peers and managers (P1, P5–7, 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 (P11) 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).

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 was 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 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 ($\sigma = 3.7$) 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



Feature	Task 1: Very useful	Task 1: Somewhat useful	Task 1: Not useful	Task 1: No basis to answer	Task 2: Very useful	Task 2: Somewhat useful	Task 2: Not useful	Task 2: No basis to answer
Gather code to new notebook	12				10	1	1	
Highlight lines code depends on	7	3	1	1	3	7	2	
Paste gathered code as cells	3	6	3		3	3	5	1
Reorder gathered code as cells	2	5	1	4	6	1		5
Paste gathered code as text	3	4	3	2	1	3	4	4
Review versions in revision browser	3	3	1	5	3	4		5
Recover deleted / overwritten cells	4	1		7	3	3	1	5

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.

(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 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).

Two analysts believed that comments, when close to the code, should be gathered alongside the code they comment on (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).

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 way 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

To help analysts manage messes in their code, we offer tool builders the following suggestions:

Support a broad set of notebook cleaning tasks. While slicing and ordering code is a key step in cleaning notebooks, analysts still need support for many other cleaning tasks. This includes refactoring code (e.g., eliminating duplicates and extracting methods), restructuring notebooks (e.g., merging cells), polishing visualizations, and providing additional documentation to explain the code and results. Many of these tasks still lack tool support in computational notebooks.

Design versioning tools to support many ways of organizing code. In our study, and in Rule's study of Janus [30], analysts used cell version histories less than expected. Is this because of issues in tool design, or because of the studies' length? Evidence from our study lends credence to both claims. Some analysts in our study told us they would not use the "Gather to Revisions" feature, as they wrote versions of code in a way that our system could not detect, i.e. duplicating and changing a cell's content elsewhere in the notebook. For future tools, two cells' "sameness" should not be determined by a cell's placement, but perhaps by using heuristics such as text similarity. Furthermore, several participants reported they didn't have enough time to create versions during the study, suggesting the need for longer programming sessions, and perhaps long-term deployments, in future studies.

Code gathering with history and compositionality. 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 should use a persistent execution log. Several participants wanted to create a cleaned notebook in a series of steps, that is, 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.

Reuse code gathering tools in other programming environments. Code gathering can be useful in other tools, such as read-eval-print loops for interpreted languages like R, Python, Scala, and others. These interpreted languages are another popular category of tools for data analysts.

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 produce 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.

ACKNOWLEDGMENTS

We thank Christian Bird and Ian Drosos for helpful discussions about tool and study design, and the data scientists at Microsoft who participated in the interviews and studies.

REFERENCES

[1] 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.* Technical Report. Stanford University.

[2] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the ACM Symposium on User Interface Software and Technology.* ACM, 473–483.

[3] Brian Burg, Andrew J. Ko, and Michael D. Ernst. 2015. Explaining Visual Changes in Web Interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology.* ACM, 259–269.

[4] 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 ACM International Conference on Management of Data.* ACM, 745–747.

[5] 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 *Proceedings of the IEEE International Conference on Software Maintenance and Evolution.* IEEE, 1–10.

[6] 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 International Conference on Intelligent User Interfaces.* ACM, 219–228.

[7] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the USENIX Workshop on the Theory and Practice of Provenance (TaPP'12).*

[8] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the ACM Symposium on User interface Software and Technology.* ACM, 91–100.

[9] Andrew Head, Elena L Glassman, Björn Hartmann, and Marti A Hearst. 2018. Interactive Extraction of Examples from Existing Code. In *Proceedings of ACM Conference on Human Factors in Computing Systems.* ACM, Article 85.

[10] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the ACM/IEEE International Conference on Software Engineering.* ACM/IEEE, 654–664.

[11] Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid Web Application Reverse Engineering via Interaction recording, Source Tracing, and Library Detection. In *Proceedings of the ACM Symposium on User Interface Software and Technology.* ACM, 270–279.

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

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

[14] Jison. http://jison.org

[15] Jupyter. http://jupyter.org/

[16] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2917–2926.

[17] Kyle Kelley and Brian Granger. 2017. Jupyter Frontends: From the Classic Jupyter Notebook to JupyterLab, nteract, and Beyond. Video. In *JupyterCon.* 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 ACM Conference on Human Factors in Computing Systems.* 1265–1276.

[19] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, 25–29.

[20] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, 147–155.

[21] 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 ACM Conference on Human Factors in Computing Systems.* ACM, Article 174.

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

[23] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. 2013. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *Proceedings of the IEEE International Conference on Software Engineering.* IEEE, 23–32.

[24] Yun Young Lee, Darko Marinov, and Ralph E. Johnson. 2015. Tempura: Temporal Dimension for IDEs. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, Vol. 1. IEEE/ACM, 212–222.

[25] 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.

[26] Emerson Murphy-Hill and Andrew P Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (2008).

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

[28] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, 105–108.

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

[30] Adam Rule. 2018. *Design and Use of Computational Notebooks.* Ph.D. Dissertation. University of California San Diego.

[31] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work and Social Computing.* ACM, Article 150.

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

[33] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the ACM Conference on Human Factors in Computing Systems.* ACM, Article 32.

[34] Francisco Servant and James A. Jones. 2012. History Slicing: Assisting Code-Evolution Tasks. In *Proceedings of the ACM International Symposium on the Foundations of Software Engineering.* ACM, Article 43.

[35] 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 ACM*

Conference on Human Factors in Computing Systems. ACM, 3509–3521.
[36] Ryo Suzuki. 2015. Interactive and Collaborative Source Code Annotation. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, Vol. 2. IEEE, 799–800.
[37] Unofficial Jupyter Notebook Extensions. https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/.
[38] Mark Weiser. 1981. Program slicing. In *Proceedings of the International Conference on Software Engineering*. IEEE, 439–449.
[39] YoungSeok Yoon and Brad A. Myers. 2012. An Exploratory Study of Backtracking Strategies Used by Developers. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 138–144.
[40] YoungSeok Yoon and Brad A. Myers. 2015. Supporting Selective Undo in a Code Editor. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, Vol. 1. IEEE/ACM, 223–233.