Notebook Archaeology: Inferring Provenance from Computational Notebooks

David Koop

Northern Illinois University, DeKalb, IL, USA
dakoop@niu.edu

Abstract. Computational notebooks allow users to persist code, results, and explanations together, making them important artifacts in understanding research. However, these notebooks often do not record the full provenance of results because steps can be repeated, reordered, or removed. This can lead to inconsistencies between what the authors found and recorded, and what others see when they attempt to examine those results. However, these notebooks do offer some clues that help us infer and understand what may have happened. This paper presents techniques to unearth patterns and develop hypotheses about how the original results were obtained. The work uses statistics from a large corpora of notebooks to build the probable provenance of a notebook's state. Results show these techniques can help others understand notebooks that may have been archived without proper preservation.

Keywords: notebook · provenance · archaeology

1 Introduction

As computational notebooks replace paper scratchpads, there are similarities to the type of work involved in understanding and utilizing these notes. While some notetakers carefully order and prepare their notes, others archive very raw notes with arrows indicating reordering, strikethroughs indicating deletions, and extra pages inserted to provide added details. These may be rewritten, but this takes time which may not be invested due to a low likelihood that the notes will be consulted. Computational notebooks exhibit similar patterns with some that are well-polished and designed for others to read and reuse, while others are scratchwork that may never be revisited. However, when one needs to reexamine past results (especially older results or those from others), these raw notes can present a challenge.

One of the more difficult situations is when a notebook contains results from multiple sessions, work from different time periods or from different contributors. Akin to having pages from multiple investigations combined together, understanding often requires not only ordering but also separating the pieces according to the different sessions. Unfortunately, there often is not a recorded history of exactly when each action was taken. Sometimes, notebooks have dated entries or pages with ordered numbering, but uncollated notebook pages crammed in a folder are not uncommon. While the format of today's computational notebooks is different, similar challenges remain. Computational notebooks can be shared for collaboration, analyses may be split in different notebooks, and some results may be outdated. The order of additions, changes, and executions, is hinted at by cell execution counts, but using these counts to surmise events is not straightforward.

Our goal of *understanding* the past is different from successfully *executing* the notebook. This goal is also important for those wishing to reuse the notebook, but we are curious about the provenance of the notebook–all the steps taken in manipulating the notebook, including the executions of individual cells. In many cases, a top-down execution strategy allows the notebook to successfully execute, but it can conflict with the *actual* execution order indicated by cell's execution counts. In addition, a successful run may still lead to different results; the results saved in a notebook may not match those generated even when the execution is successful [20, 27].

This paper uses collections of notebooks along with notebook session histories to build an understanding of common patterns in notebook use. From this information, we construct an algorithm that fills in gaps of its execution provenance using the breadcrumbs a notebook provides. For example, the common practice, reinforced by the interface, of executing cells in consecutive order helps us fill in gaps in the provenance indicated by the saved notebook cell positions and execution counts. Our inferred provenance is necessarily more uniform that the actual provenance because some operations cannot be derived from only saved notebooks; we cannot determine if a cell was moved to a different location. In addition, where cells have changed or been deleted, we must project the provenance onto the current state of the notebook.

To infer provenance, we use a corpus of notebooks and a separate collection of histories of executed code from notebooks. In addition to looking to model user interactions with notebooks, we find some interesting results showing users commonly revisit and reexecute notebooks across multiple sessions and some differences between how users structure their code. To evaluate potential provenance inference algorithms, we use static code analysis to highlight dependencies that are or are not satisfied in the constructed provenance. The many difficulties we found suggests notebooks would benefit from improved provenance tracking. At the same time, the ability to produce plausible provenance from the limited information can be useful in better understanding the millions of already published notebooks.

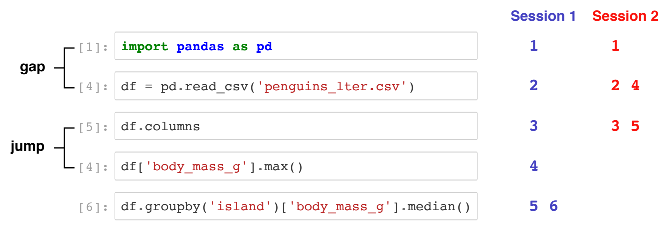
2 Related Work

A *computational notebook* is a sequence of code and text blocks called cells. Generally, a user executes individual code cells one at a time, going back to edit and re-execute cells as desired. This is in contrast to scripts where all code is executed at once. In addition, new cells may be later inserted between existing, already-computed cells, so it possible that the semantics of a variable change. There exist a variety of different computational notebook environments [9, 28, 24, 1, 2, 16, 17], all of which use text and code cells with computational results shown inline. Generally, these environments serve to mimic paper notebooks that document a scientist's work and include text, computations, and visualizations. Notebooks persist *input code, output results, and explanatory text*, providing a single record of an analysis and any discoveries. This contrasts with other computing where source code, outputs, and explanation are stored in separate documents.

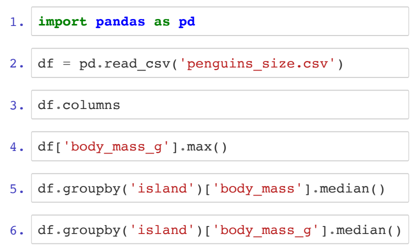
Despite this encapsulation of research artifacts, the reproducibility of notebook results has drawn considerable concern. Recent studies on the reproducibility of notebooks provide evidence that current practices fall short; even for those notebooks where dependencies are specified and cell order is unambiguous, hidden dependencies and out-of-order execution can hinder reproducibility [20, 27]. Work has also been done to help diagnose non-reproducible notebooks and reconstruct execution schemes by examining the dependencies between code cells [27]. Other solutions seek to modify the execution semantics of notebooks in order to improve reproducibility in the future. Nodebook [15] and Datalore [5] enforce in-order execution semantics on notebooks, and reactivepy restricts cells to single definitions to allow reactive execution [21]. Dataflow notebooks make dependencies between cells clearer, allowing the system to reactively update dependent cells as well as determine when cells are stale or up-to-date [13]. NBSafety uses static analysis techniques to highlight cells that may be stale, helping users see the effects of code changes without modifying the normal interaction or execution in notebooks [14]. Other research has shown that around 1 in 13 cells are duplicated in notebooks [12]. While provenance may also aid in reproducing results, our goal is different–to infer history from saved notebooks.

There has also been study of problems with the current modes of use in notebooks [4], and there has been work to improve usability. In particular, techniques help users better understand navigation of the existing notebook structure, something which can aid in inferring provenance. For messy notebooks, techniques have been developed to fold blocks of cells [22] or help users gather only those cells germane to a particular artifact [7]. It can also be important for users to understand how their actions affect the evolution of a notebook, and interfaces that present such information augment users' memories [10, 11]. In real-time collaboration settings where users are working on a shared notebook, users tend to require some level of coordination, and understanding other users' contributions is often complicated by the non-linear structure of notebook work [26].

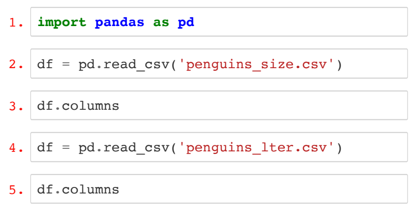
There are a number of solutions for tracking provenance in scripts [19], and some specific features and work to address provenance in notebooks. IPython tracks the history of all code that was run in a session in a user-level SQLite database [18]. This history is available in a notebook to document the provenance of executed code, but it is not stored with the notebook. Jupyter [9] also creates checkpoints that keep snapshots of a notebook through time, although these are generally overwritten. Other opportunities to improve the provenance of notebooks includes storing the provenance directly with notebook results [25].


```
[1]: import pandas as pd
[4]: df = pd.read_csv('penguins_lter.csv')
[5]: df.columns
[4]: df['body_mass_g'].max()
[6]: df.groupby('island')['body_mass_g'].median()
```

(a) Notebook


```
1. import pandas as pd
2. df = pd.read_csv('penguins_size.csv')
3. df.columns
4. df['body_mass_g'].max()
5. df.groupby('island')['body_mass'].median()
6. df.groupby('island')['body_mass_g'].median()
```

(b) Session 1 History


```
1. import pandas as pd
2. df = pd.read_csv('penguins_size.csv')
3. df.columns
4. df = pd.read_csv('penguins_lter.csv')
5. df.columns
```

(c) Session 2 History

Fig. 1: The notebook records the final state, after edits to cells, while the session histories record the code at each cell execution.

In many settings including with workflows, there has been work to infer provenance, whether that be to improve its granularity [3] or its precision [6]. ProvenanceCurious infers data provenance from annotated scripts, using the abstract syntax tree to build the provenance graph [8]. Our work deals with less well-defined data in that we seek to infer likely provenance given limited information, knowing much can be missing, about the final notebook state including cell positions and execution counts.

3 Definitions

A *computational notebook* is a sequence of code and text blocks called *cells* (see Fig. 1a). A *code cell* contains any number of lines of executable code, while a *markdown cell* contains text that is often explanatory and rendered from markdown syntax. Our work will focus on Jupyter notebooks [9] written in IPython [18], but many of the concepts and ideas will translate to other systems. In Jupyter, cells are ordered by *position* which indicates their location in the notebook; we can associate a numeric index to track this (one at the top, increasing down the page). There are a number of ways a user can modify a notebook; a user may add a new cell, delete cells, move cells to new position, edit a cell, or execute cells. Any operations involving multiple cells can be decomposed into operations on a single cell. Other more complex operations like copy-and-paste can similarly be decomposed into delete/add/edit operation chains. Note that many of these operations modify the structure of a notebook in a way that can cause headaches in inferring past provenance.

Jupyter uses a web-based front-end to facilitate editing and execution, but the actual computation is done by a back-end *kernel*. Different kernels exist for various programming languages, but we will concentrate on Python, the most commonly used with Jupyter. When Jupyter creates a new connection between the notebook and the kernel, a new *session* begins. Sessions are closely linked to the kernel, tracking the code that is run and the outputs that are generated. Each session has a global counter that is used to tag code, cells, and outputs, upon a cell execution. The notebook records this number for the executed cell as its *execution count*, and the *session history* separately records the code and count in a database (see Fig. 1) This counter is reset to one each time a new session begins, meaning the same execution count can appear for different cells in the same notebook. In addition, a cell's execution count is overwritten any time it is executed, and can be deleted completely if the notebook's outputs are cleared. The execution count hints at a global *execution order*: provided each session is assigned a monotonically increasing identifier, the session identifier plus the execution count provides a global timestamp for all cells in a notebook. Unfortunately, session identifiers are not recorded, leading to ambiguity of the existing execution counts.

Then, an execution count k is *missing* when no cell has such an execution count and there exists a cell with execution count $\ell > k$. A *gap* (also known as a skip [20]) is a consecutive sequence of missing counts, and thus has a *length* (see Fig. 1a). When two cells have been executed in order, the signed difference between the positions of the first and second cells is the *jump* (see Fig. 1a). No meaningful jump occurs when this value is 1, signifying the cells were executed in a top-down manner without the user refocusing on a different cell. Putting these together, we define a *gap-jump* when we have both a gap and a jump. More precisely, given two cells with positions i and j and execution counts k and ℓ, respectively, such that there is no cell in the notebook with an execution count m, $k < m < \ell$, the *gap-jump* measure is a tuple $(\ell - k, j - i)$. In a top-down execution, $\ell = k + 1$ and $j = i + 1$, leading to a gap-jump measure of (1,1). A gap-jump of (2,1) often arises when a user executes the same cell twice, for example, after fixing a typo. A gap-jump of (1,2) might indicate a user skipping the execution of a cell in a later session. Note that these quantities represent differences so similar gap-jumps can occur in different times and locations for different notebooks.

3.1 Provenance

The *provenance* of a notebook is the sequence of all cell actions–the ordered steps that led to the notebook's state. While there is more state information that is stored with the notebook (which cells are collapsed, whether output exists, other cell metadata), we will ignore those because they do not affect the order or execution of cells. In general, this provenance information is not stored in the notebook, meaning for the majority of notebooks that do not use some versioning scheme as an extension to Jupyter, we do not know what this provenance is. The remainder of this paper seeks to present information that is useful in analyzing notebooks and solutions that use this data to infer potential provenance.

The *execution provenance* of a notebook is the chronological record of code cell executions, that is each cell (including its code and position) that was executed. Note that markdown cells and unexecuted code cells are omitted from this provenance. This history matches what IPython records in its history.sqlite database, but is not stored with the notebook in most cases. Specifically, when a cell is executed twice but was edited in between executions, the execution provenance records the cell code for each execution. Because the notebook only stores the most recent edit to each cell, this full provenance is impossible to infer. Instead, we will will focus on the *projected* execution provenance which substitutes the "closest" cell to stand-in for the state of the cell executed in the past. For a cell that was edited, the closest cell remains that same cell, regardless of where it was moved to. For a cell that was deleted, the closest cell could be any, either one that fits well into the sequence or simply the cell that was executed after it. Recall that only executed cells figure into this provenance. With the projected provenance, we can re-execute the notebook in a manner that approximates the original execution. With no reexecutions or deletes, the projected provenance is the same as the original execution provenance. Given a saved, executed notebook, our goal is to infer this projected execution provenance.

4 Data and Statistics

Because we are inferring the provenance, we need to make decisions about how a notebook was *likely* executed based only on its final state. To do this, we will lean on a corpus of notebooks as well as one of session history collected from the GitHub online repository. With a diverse set of notebooks, we can gain an understanding of the distribution of various notebook features like gaps and jumps, and the session histories provide more detail on how cells are modified and re-executed. This will allow us to derive some general patterns related to notebook use, reuse, and editing. These will be used to inform an algorithm that seeks to infer the projected provenance.

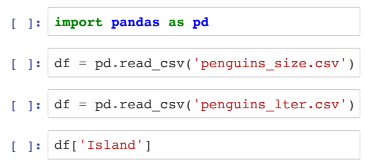
4.1 Notebooks

A number of studies have harvested notebooks from GitHub for research [20, 23], and these notebooks have shown significant diversity ranging from polished, explanatory documents to single-use scratchwork; ranging from a handful of cells to hundreds; and ranging from programming cheat sheets to in-depth machine learning experiments. We have employed similar strategies to existing work, obtaining new notebooks by querying GitHub for the distinguishing `.ipynb` file extension. From a corpus of millions of notebooks collected through February 2021, we randomly sampled 100,000 notebooks. To focus on IPython, and in particular notebooks that use Python 3, we filtered notebooks based on that metadata. Checking that notebooks that could be meaningfully loaded, we were left with 65,119 notebooks, and of these, 58,276 have at least one executed cell.

Gaps & Jumps The easy case for inferring provenance is when we have no gaps or jumps in a notebook. In this case, the cells have been executed in positional order–that is from the top to the bottom of the notebook. There were 17,587 notebooks (30.18%) that fell into this category, something achieved by running all cells consecutively in a notebook. In the remaining notebooks, we may have some that involve multiple sessions. We know we have a notebook that has been used in multiple sessions when the same execution count appears twice; this is sufficient but not necessary. For those notebooks assumed to be single-session, we can count the number of gaps and jumps, and gap-jumps. The standard execution of two cells in top-down order, a gap of 1 and jump of 1 is by far the most common, occurring almost 80% of the time. Another 10% have larger gaps but no jump, likely indicating repeated execution of the second cell. About 3% of those pairs with a larger gap have actual jumps.

Sessions An important issue in understanding how the execution counts relate to the provenance of a notebook lies in how many sessions a notebook has been used in. Recall that the execution count restarts at one in each new session so we cannot estimate how many cells may have been executed in total without first knowing how many sessions there were. We can conservatively estimate this number by finding the maximum number of repeats of an execution count in a notebook. Based on this cardinality, most (87.56%) of the notebooks have only a single *detectable* session, 10.06% have at least two sessions, and 1.66% have at least three. We can also compute a lower bound for the number of cells that must have been executed by summing the maximum execution count for each number of repeats. For example, consider a notebook with execution counts $[1, 6, 4, 5, 2, 4, 6, 1, 2, 3, 4]$. Because we have three 4s, and two 6s, the notebook had at least $4 + 6 + 6 = 16$ executed cells; the last 6 is added because it is the maximum execution count among those counts that appear at least once. Calculating this lower bound for all notebooks, there is one notebook where at least 10,916 cells were executed, but the median is 27. Note that this is usually *more* than the number of cells in the notebook. To that end, we can examine the ratio of executed cells in the notebook versus our lower bound of those executed. Using the example data, this ratio is 11/16 0.69. Interestingly, the interquartile range of this ratio is wide, from 0.15 to 0.89, indicating different modes of interaction in notebooks. Low ratios indicate many reruns of cells while higher ratios may indicate single executions or a session where the notebook was reexecuted.

Static Code Dependencies In addition to the cell positions and execution counts, we have the code of each cell. While this work does not seek to execute the cells, we can employ static code analysis to inform our understanding of relationships between cells, similar to techniques used by Osiris [27] and NBSafety [14]. Because it is a dynamic language, it more difficult to statically analyze Python code, but we can make some progress in investigating dependencies between cells. Specifically, the language makes it possible to differentiate between definitions and references of a particular identifier (or name). We care about those



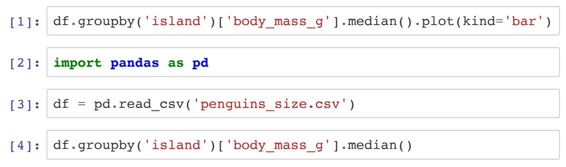
```
[ ]: import pandas as pd

[ ]: df = pd.read_csv('penguins_size.csv')

[ ]: df = pd.read_csv('penguins_lter.csv')

[ ]: df['Island']
```

(a) Dependencies



```
[1]: df.groupby('island')['body_mass_g'].median().plot(kind='bar')

[2]: import pandas as pd

[3]: df = pd.read_csv('penguins_size.csv')

[4]: df.groupby('island')['body_mass_g'].median()
```

(b) Ordering Issues

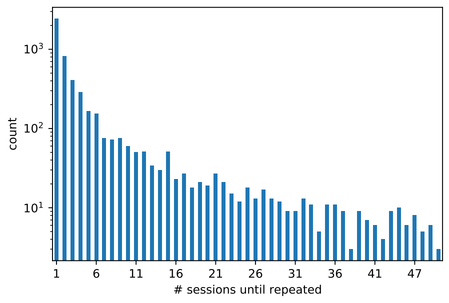
Fig. 2: In (a), the last cell is *dependent* on an earlier cell that defines `df`. Because there are two such cells, the dependency is *ambiguous*. In (b), suppose the cells are executed in positional order. Then, the first cell has *unbound* symbols (`df`) that are later defined, meaning the cell is *out of order*.

definitions that are made in one cell and then referenced in a different cell because this indicates a dependency between cells that can be used as a partial ordering. It is not foolproof as there are many potential ways to influence the global namespace, but this should cover most common cases. This also allows us to determine when cell references are potentially ambiguous–that is when two different cells assign to or define a particular name. When a third cell references this name, it is possible that reference is to either of the cells.

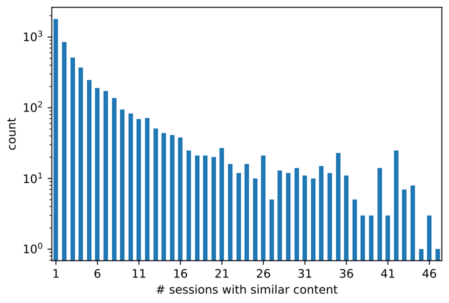
Function implementations, which do not access particular names until executed present challenges because a name in a global namespace need not be defined when the function is defined but must exist when the function is run. Because of this, it is possible that another cell defines the global after the function, but this is still valid because the function is not executed until after that definition.

We define four types of symbol dependencies which also lead to potential relationships between cells. Specifically, any code cell that has a referenced symbol that was not first defined in that cell has a likely dependency on another cell. In this case, we call the symbol and cell *dependent*. When that referenced symbol is defined/assigned in more than one cell, the symbol or cell is *ambiguously dependent* (see Fig. 2a). These two definitions are unrelated to the order cells were executed in. The dependency, ambiguous or otherwise, exists regardless of any specified execution order. We found that most notebooks (94.76%) had at least one dependency, although more than half (53.58%) had zero ambiguous dependencies. Among those with ambiguous dependencies, the average ratio of ambiguously dependent cells to total cells was about a quarter (25.49%). Again, this shows different variable definition patterns, hinting that some users may be keenly aware of issues with defining a variable more than once.

The second pair of definitions are related to execution order, meaning we can use these to evaluate inferred execution provenance. Given an ordering of cells, when a cell references a symbol that has not been defined/assigned in a cell earlier in the ordering, we have a *unbound* symbol. (We specifically exclude builtin symbols in these calculations.) This may mean the symbol is later defined, or it may mean the symbol is *never* defined. There also may be cases where the symbol would be added to the namespace by a wildcard import or some other



(a) Sessions Until Repeat



(b) Number of Repeated Sessions

Fig. 3: In the session history, we find evidence that users revisit notebooks. Usually, this is soon after the previous session (a), and many notebooks are revisited many times (b).

code. The subset of those symbols that are later defined are the classified as *out-of-order* (see Fig. 2b). The number of out-of-order cells will serve as a metric to evaluating how well our provenance inference technique works.

4.2 History

Inspired by Macke et al.'s use of IPython session history as a proxy for user behavior [14], we take a similar approach to understand execution patterns. Note that this assumes that the mode of interaction is via notebook, but it is possible this history is recorded from console-based IPython interactions as well. We will assume the code represents input from cells. Because the session history contains all executed code, we have a more complete record of executed cells, and can better see how often and when cells are re-executed. The patterns from these histories will help us determine when to infer repeated executions in our provenance reconstruction.

We downloaded all history.sqlite files from GitHub, and found 570 unique files with 86,711 sessions, many of which were empty. We were were able to extract code from 43,529 sessions. We eliminated sessions with 10 or fewer lines (27,328) and those with 100 or more (2,058), the latter due to the number of possible checks required. In the 14,143 sessions, we found 977,728 "cells". (IPython calls these lines despite many being composed of multiple lines, but cells of notebooks become lines in the history database.)

Because we were not concerned about the actual execution of the cells, errors or outdated code are fine. Our goal is to find all repeated executions. To do so, we test all combinations of cells in the same session for similarity. Following Macke et al., we classify two strings as similar if the Levenshtein distance between them, normalized by dividing by the maximum length of the two strings, is less than 0.2, that is roughly 80% of the code is the same [14]. Running this repeat detection across all sessions, we found 222,202 likely repeated cells. Note that this does not mean that all of these cells were (modified and) re-executed as there may also be duplicate cells [12].

Importantly, this allows us to estimate the probability of a (1,0) gap-jump, that is a change in execution count of 1 (no real gap), and a jump of 0, staying in the same place. Recall that we have no measure of this from the notebooks because there could be no data about jumps of 0 as the re-execution would overwrite the previous cell's execution count. The next cell matches the previous cell in approximately 10% of all repeats (102,580 of 1,024,508). This is out of 447,244 cells analyzed (for 23% of all executions); the number of repeats exceeds the number of cells because we count all pairs of repeats. We will use this probability of repeating cells to guide decisions about how to fill in the missing gaps.

We can also look for repeats *across* sessions; this shows how often a notebook was revisited in a later session. To accomplish this, we first de-duplicate the individual sessions, leaving only one copy of each group of repeats. Then, we compare earlier sessions with later ones. If the later session repeats at least 50% of the cells from the earlier session (repeats measured via the same Levenshtein distance criteria), we classify it as a revisit of of the notebook. If, at any time after comparing ten lines from the first session, we have less than 10% overlap, we quit checking for overlaps. We only searched for repeats within 50 sessions due to computational time, but found 5,163 sessions repeated, some multiple times. The results show that most repeats occur quickly, often within the first 10 sessions (see Fig. 3a). In addition, many sessions are repeated multiple times with a significant number of notebooks being revisited over 20 times (see Fig. 3b).

5 Algorithm

Our goal is to infer projected execution provenance, the order in which we should execute the notebook's current cells to best emulate all of the past executions of notebook cells. Again, we will use the position and execution counts to guide decisions, but also the patterns and frequencies that were gathered from the GitHub notebooks and session data. Because multiple sessions introduce added complexity, we will be begin by examining the single-session case, and then discuss how this can be extended for the multi-session case.

Worst Case Note that even with the best algorithm, it is possible to have wildly different actual provenance than that inferred by this model. For example, each cell could be executed repeatedly in a separate session, allowing any possible execution count that has no relationship to the other cells in the notebook. They may also be moved to reorder their positions. Given the history data, this is likely an extremely rare occurrence, but it is possible, as with an archaeological dig, that the most likely explanation given all evidence is not correct. Someone trying to deliberately obfuscate the provenance of a notebook can very likely succeed.

5.1 Base Algorithm

The base algorithm strictly follows the order implied by the execution counts. We don't know what happens with any gaps so we can fill them in with repeated

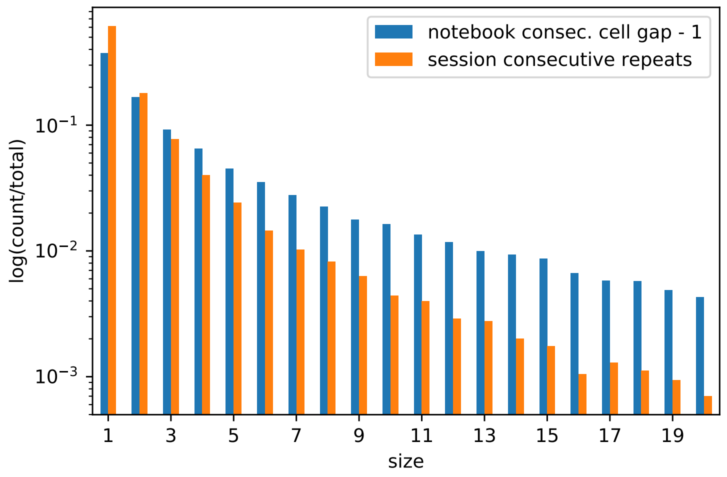


Fig. 4: Comparing the gap ($\geq$ 2) between two adjoining cells in notebooks to the number of repeats of the same code in session history. A gap of n can be caused by $n-1$ repeats of the second cell.

executions of the cell at the end of the gap. While this seems reasonable, and we will see a nice extension to the multi-session case, the results (see Section 6) show this to be a somewhat poor strategy, as many symbol definition-reference pairs are out of order. The reason lies with the probability of repeated execution being *lower* than that of the standard (1,1) gap-jump. For example, if Fig. 1 showed execution counts [1,6,7,4,5], it is most plausible that the user executed all five cells in order and then went back to execute the second and third cell again. The base algorithm would instead jump from the first cell to the second-to-last, likely missing declarations or computations.

Extension to Multiple Sessions This base algorithm does, however, provide a nice path to determine how to segment a multi-session notebook. Recall that we can obtain a lower-bound on the number of sessions by finding the count of the mode (the maximum number of repeats for an execution count). Given this, we can start from an alignment of all the sessions at this mode, and use a greedy approach to find the best next step in each session. Specifically, given the next possible step, we look up the relative frequencies of the induced gap-jump, picking the highest one. The step can be in either direction (to the next lower or higher execution count) as the process is the same. Continuing this process allows us to assign each cell to a session, at the same time building the projected provenance. Note that the order of the sessions is difficult to ascertain because there is nothing in this algorithm that considers when executed cells may have been reexecuted–this is only implicitly captured in the gap-jump data.

5.2 Informed Algorithm

In the single session case where we assume the entire notebook state was generated in one session, any gaps mean that we have executed a cell twice or executed a now-deleted cell. Since re-executing a cell often does not change state, we will assume that all gaps are re-executions, rather than deleted cells. (Any code that modifies a variable based on its current value may be problematic here; it may be possible to flag such cells.) Then, the problem is finding the best path through cells that have execution counts *higher* than the the specified number. Here, we can leverage the distribution of jumps, but we must also consider the endpoint of the gap, the cell we must end at. In addition, the user may have jumped to the final cell and re-executed it as many times as necessary.

From the data from the session history database, we know that both consecutive cell executions and immediate re-executions appear frequently. In addition, if we see two cells that adjoin in position, and the cells are also consecutive by execution count, the gap between those cells may often be caused by multiple executions of the second cell. This may come about because of trial-and-error in getting the code correctly by fixing typos, modifying parameters, etc. Figure 4 shows that the distribution of these gaps is similar to the distribution of sessions where the same line is executed multiple times in succession. The single repeats in sessions make up a greater proportion of repeats than (2,1) gap-jumps in notebooks, but then decline faster, meaning long length-m repeats are *less common* than $(m-1 \gg 2,\ 1)$ gap-jumps. This may be explained by the idea that the gaps can also be caused by jumping to other parts of the notebook that are later re-executed or removed, and it may also be more likely that cells tried once or twice are later deleted.

Our goal is to come up with likely execution provenance to fill in the gaps. From the notebook data, we know that a (1,1) gap-jump is most likely. In addition, the frequency of jumps of 1–that the next cell in execution count is also the next cell positionally, is about 9 out of 10 (89.56%). Thus, we expect mostly top-down order (and without other information are best to assume this), but this must mesh with the notebook. A negative jump—executing a cell earlier in the notebook, requires a negative jump at some point in the execution provenance (since we are ignoring moves). We can maximize the number of consecutive executions as cell numbering permits, but need to make at least one jump. Thus, our best strategy may be to move forward as much as possible with a single jump. Since this may still not cover the entire gap, we can repeat the final cell until reaching the desired execution count.

More formally, we wish to determine the projected execution provenance for the gap defined by cells c_i and c_j which have positions i and j and $\text{count}(c_j) - \text{count}(c_i) > 1$. We will attempt to use all cells that come after c_i or before c_j *and* have execution counts greater than c_j's. Formally, we want cells $\{c_k\}$ with $\text{count}(c_k) > \text{count}(c_j)$ and either $k < j$ or $k > i$. If the size of candidates, $\{c_k\}$, is greater than the size of the gap, we use those positionally before c_j first and then those after c_i (the jump happens earlier). If there are too few candidates, we choose to repeat c_j as many times as necessary, drawing on the session history data showing repeats to be a common occurrence. In between those executions, we will need to jump if the cells are positionally out-of-order, $j < i$, or if the gap is smaller than the jump, $j - i > \text{count}(c_j) - \text{count}(c_i)$. See Algorithm 1 for

Algorithm 1 Informed Provenance Algorithm

```
function FillProvenance($C$)
    $order \leftarrow []$                                          ▷ initialize result
    $C \leftarrow \{\text{Cell}(pos = 0, count = 0)\} \cup C$      ▷ add dummy cell
    sortByPosition($C$)                                            ▷ $c_i$ is at position $i$
    for $c_i, c_j \in$ paired(sortedByCount($C$)) do
        $gap \leftarrow \text{count}(c_j) - \text{count}(c_i)$
        if $gap = 1$ then
            $order$.append($c_j$)
        else
            $n = \min_{s \geq 0}\{s \mid \forall k \in [s, j]\ \ \text{count}(c_k) \geq \text{count}(c_j)\}$
            $m = \max_{t \leq |C|}\{t \mid \forall \ell \in [i, t]\ \ \text{count}(c_\ell) \geq \text{count}(c_j)\}$
            if $j > i$ then
                $posGap \leftarrow \min(j - i - 1, gap - 1)$
            else
                $posGap \leftarrow gap - 1$
            $numBefore \leftarrow \min(posGap, j - n)$
            $numAfter \leftarrow \min(posGap - numBefore, m - i)$
            $numRepeats \leftarrow posGap - numBefore - numAfter$
            for $k \in i + 1, i + numAfter$ do
                $order$.append($c_k$)
            for $k \in j - numBefore, j - 1$ do
                $order$.append($c_k$)
            for $\ell \in 1, numRepeats + 1$ do                    ▷ always do once so $c_j$ is added
                $order$.append($c_j$)
    return $order$
```

details, and Fig. 5 for an example showing how a gap in the provenance is filled by the algorithm.

Extension to Multiple Sessions The extension to the multiple session case brings the possibility of having a cell with a *lower* execution count being run in a previous session at a higher execution count. We can attack this by first assigning each cell to a session, but then we have to find a way to order the sessions so that we know which cells will eventually be re-run (and their execution count overwritten). One option here is to assume that later sessions will be more contiguous. Assuming this can be solved, the algorithm continues as in the single session case, allowing gaps to be filled by inferring executions of not only cells whose execution counts are greater but also cells in later sessions. A second option is to thread this with the cell-session assignments, as in the base algorithm. Here, we will rank assignments of cells to sessions based on the provenance they induce. For example, in Fig. 1, assigning the third and fourth cells to the same session forces less frequent jumps than assigning the second and third to the same session instead (as was the case).

variables		i			m			n	j		
pos	···	3	4	5	6	7	8	9	10	11	···
count	···	13	22	23	24	7	8	21	19	20	···
inferred	···	13	14 22	15 23	16 24	7	8	17 21	18 19	20	···

Fig. 5: Filling in the gap between c_3 and c_{10}. These cells are the third and tenth in the notebook and have execution counts of 13 and 19, respectively. This means $gap = 19 - 13 = 6$, and we need to decide which five cells were executed during steps 14 through 18. The algorithm assigns counts to cells with execution counts higher than 19 following c_3 ($numAfter = 3$) and preceding c_{10} ($numBefore = 1$), and then assigns c_{10} to repeat once ($numRepeats = 1$). The inferred execution order shows that some cells (e.g. c_4) are executed more than once.

6 Evaluation

We used the code dependency measures defined in Section 4 to evaluate how well our provenance captures a realistic execution history without attempting to run the code. Other work has attempted to reexecute entire notebooks, classifying them as reproducible when the execution succeeds and the results match. This work is subject to a number of variables including data availability, library and package dependencies, and execution order [20, 27]. Our work focuses on execution order in a way that is agnostic to the results of the code. We lean on the static code analysis to find those identifiers that are defined out of order in order to test our approach. Again, this analysis looks at the names that appear across cells. Any name that is referenced before it is introduced would be out of order.

We choose to compare three different techniques using the code dependency metrics. First, we take the top-down approach–simply execute all executed cells in positional order. Second, we examine the base approach that goes not by position but by the stored execution counts. Finally, we compare with the informed algorithm that attempts to fill in gaps with other cell executions.

There were 1,519,914 cells in the notebooks. The top-down approach has only 2,196 out-of-order cells, compared with 39,386 for the base algorithm, and 26,619 for the advanced algorithm. The top-down median number of out-of-order cells per notebook was 1 compared with 2 for the other approaches. Perhaps surprisingly, both algorithms do significantly worse as measured by out-of-order executions than top-down execution. Note, however, that the single-session advanced algorithm does improve significantly on the base algorithm. Reflecting on this further, the actual execution is often messier than the top-down order, and may result in more similar results.

One notebook that has problems with out-of-order cells under the inferred provenance has definitions (imports) as the first cell but an execution count of 141, followed by a second cell with count 132, before cells with the count sequence 2, 3, 4, 5. The algorithm assumes that the second cell was the first execution (1), leaving the actual first cell until much later. Most likely, this second cell was inserted later, something that would be difficult to determine without other information. Around 100 notebooks have fewer out-of-order cells using our algorithm than the top-down execution. One of these has a cell with a plot as output that was executed last yet featured as the first cell. Again, the cell may have been moved, but following the execution counts here provides what is likely more accurate provenance.

7 Discussion

The data we have gathered helps shine a light on patterns of notebook interaction, allowing us to infer the provenance of a particular notebook. However, there are several limitations. First, because we cannot determine if a cell was removed or a cell was added or moved, our inferred provenance necessarily lacks some of the actions that a user might take. Second, we do not have data that links session histories with notebooks. The session histories do not record notebook locations or filenames, and it is unclear how to effectively link the session histories on GitHub to existing notebooks. The histories are maintained in a separate area of the filesystem (a "dot" directory in a user's home directory), and that is generally not included in the same repository as published notebooks, if any published notebooks exist.

Improving Evaluation A true recording of the provenance would be beneficial in better profiling notebooks. This would help better tie data from notebooks and session histories together, as well. Extensions that version notebooks and their cells are very useful for this purpose [11], but the vast majority of notebooks lack this information. Another opportunity may be those who version their notebooks using conventional tools like git. Even though these will lack the granularity of a system that tracks all operations on a notebook, they would allow improved inference of changes in the notebook as adds, moves, and deletions may be more effectively estimated.

Using All Cells Notebooks contain more than code cells, but we have restricted most of the discussion of provenance to code. Literate programming emphasizes a combination of code with text, and this text is included in markdown cells in Jupyter. While some notebooks have more text than others, for those that do, we may be able to use information about the position of this text to determine logical sections of a notebook which may aid is session partitioning. In addition, the calculations of gaps and jumps ignore markdown cells, but having such cells in between may affect the probability of a particular repeated execution or jump, thus improving the algorithm.

Using Code Dependencies Both Osiris and NBSafety look to static code dependencies in order to derive more likely execution sequences and flag stale cells, respectively. We instead use these code dependencies to evaluate provenance reconstructed via statistical trends of gaps, jumps, and repeats. We expect that code dependencies can be used to improve this provenance construction, but if we optimize for that, we lose our ability to evaluate the proposed algorithm. A possible solution is to use execution of the notebooks, comparing output values directly. This is prone to the other issues mentioned earlier, including missing data and dependency issues, but may provide enough results to judge the efficacy of a hybrid solution.

Determining Sessions Our method to determine sessions looks reasonable in that it segments the notebook in meaningful pieces most of the time. However, it induces more issues with out-of-order cells than top-down execution. This is not totally unexpected, as we expect most users to execute cells in top-down fashion so cell execution counts that are out of order are actually more likely to have been reexecuted than simply executed in a random fashion. Thus, when execution counts are missing, it is actually more likely that the cell was repeated (or potentially moved) than it was executed in a haphazard fashion.

Localized Predictions Our model for inferring provenance uses global distributions for guidance, but different users have different approaches to notebook use [23]. Those with few but lengthy cells that function more like scripts will be editing and re-editing single cells over and over while those with many shorter cells will likely be executing cells in sequence more often. In addition, even among the same users, notebooks used for exploration may be structured differently than those used for explanation. It may be useful, then, to classify notebooks or users according to particular styles in order to better infer provenance. We also expect that in later sessions, many of the cells have fewer immediate re-executions. Often, the first execution of a cell raises an exception due to a typo, a missed import, or some flawed logic, leading a user to correct that problem. Thus, some repeats may be less likely if we know the code was executed in a previous session. Another opportunity for improving predictions is understanding the content of the cell; cells importing dependencies may be executed and updated more often and with a greater probability of a jump.

8 Conclusion

We have presented methods to infer provenance from static notebooks based on knowledge gained from examining the large corpora of notebooks and session histories. These methods take a step forward in the very difficult problem of meaningfully understanding how a user interacted with a notebook. A future direction is to examine how well this computed provenance meshes with the actual results.

While we present some evidence that provenance can be inferred, the frequent ambiguities point to a need for improved provenance in notebooks. While it has been shown that many notebooks have reproducibility issues, this paper demonstrates that even with further analysis of partially-known steps rooted in statistical analysis of notebook and session data, there is not enough data to provide the type of provenance that would enable greater understanding of how a user arrived at particular conclusions and where they may have changed course. Since there are millions of notebooks that already exist, this work addresses the challenges from the past while prompting action for the future.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant SBE-2022443.

References

1. Apache Zeppelin. http://zeppelin.apache.org
2. Beaker Notebook. http://beakernotebook.com
3. Bowers, S., McPhillips, T., Ludäscher, B.: Declarative rules for inferring fine-grained data provenance from scientific workflow execution traces. In: International Provenance and Annotation Workshop. pp. 82–96. Springer (2012)
4. Chattopadhyay, S., Prasad, I., Henley, A.Z., Sarma, A., Barik, T.: What's wrong with computational notebooks? pain points, needs, and design opportunities. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–12 (2020)
5. Datalore, https://datalore.jetbrains.com
6. Dey, S., Belhajjame, K., Koop, D., Song, T., Missier, P., Ludäscher, B.: UP & DOWN: Improving provenance precision by combining workflow-and trace-level information. In: 6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014) (2014)
7. Head, A., Hohman, F., Barik, T., Drucker, S.M., DeLine, R.: Managing messes in computational notebooks. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. p. 270. ACM (2019)
8. Huq, M.R., Apers, P.M., Wombacher, A.: Provenancecurious: a tool to infer data provenance from scripts. In: Proceedings of the 16th International Conference on Extending Database Technology. pp. 765–768 (2013)
9. Jupyter, http://jupyter.org
10. Kery, M.B., Myers, B.A.: Interactions for untangling messy history in a computational notebook. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 147–155 (Oct 2018). https://doi.org/10.1109/VLHCC.2018.8506576
11. Kery, M.B., John, B.E., O'Flaherty, P., Horvath, A., Myers, B.A.: Towards effective foraging by data scientists to find past analysis choices. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. pp. 92:1–92:13. CHI '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3290605.3300322, http://doi.acm.org/10.1145/3290605.3300322
12. Koenzen, A.P., Ernst, N.A., Storey, M.A.D.: Code duplication and reuse in jupyter notebooks. In: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 1–9. IEEE (2020)
13. Koop, D., Patel, J.: Dataflow notebooks: Encoding and tracking dependencies of cells. In: 9th Workshop on the Theory and Practice of Provenance (TaPP 2017) (2017)
14. Macke, S., Gong, H., Lee, D.J.L., Head, A., Xin, D., Parameswaran, A.: Fine-grained lineage for safer notebook interactions. Proc. VLDB Endow. **14**(6) (2021)
15. Nodebook, https://github.com/stitchfix/nodebook
16. North, S., Scheidegger, C., Urbanek, S., Woodhull, G.: Collaborative visual analysis with rcloud. In: Visual Analytics Science and Technology (VAST), 2015 IEEE Conference on. pp. 25–32. IEEE (2015)
17. Observable, https://observablehq.com
18. Pérez, F., Granger, B.E.: Ipython: a system for interactive scientific computing. Computing in science & engineering **9**(3), 21–29 (2007)
19. Pimentel, J.F., Freire, J., Murta, L., Braganholo, V.: A survey on collecting, managing, and analyzing provenance from scripts. ACM Computing Surveys (CSUR) **52**(3), 1–38 (2019)

20. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: A large-scale study about quality and reproducibility of jupyter notebooks. In: Proceedings of the 16th International Conference on Mining Software Repositories. pp. 507–517. IEEE Press (2019)
21. reactivepy, https://github.com/jupytercalpoly/reactivepy
22. Rule, A., Drosos, I., Tabard, A., Hollan, J.D.: Aiding collaborative reuse of computational notebooks with annotated cell folding. Proceedings of the ACM on Human-Computer Interaction **2**(CSCW), 150 (2018)
23. Rule, A., Tabard, A., Hollan, J.D.: Exploration and explanation in computational notebooks. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 32:1–32:12. CHI '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3173574.3173606, http://doi.acm.org/10.1145/3173574.3173606
24. Sage Developers: SageMath, the Sage Mathematics Software System (2017), http://www.sagemath.org
25. Samuel, S., König-Ries, B.: Provbook: Provenance-based semantic enrichment of interactive notebooks for reproducibility. In: International Semantic Web Conference (P&D/Industry/BlueSky) (2018)
26. Wang, A.Y., Mittal, A., Brooks, C., Oney, S.: How data scientists use computational notebooks for real-time collaboration. Proceedings of the ACM on Human-Computer Interaction **3**(CSCW), 39 (2019)
27. Wang, J., Tzu-Yang, K., Li, L., Zeller, A.: Assessing and restoring reproducibility of jupyter notebooks. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 138–149. IEEE (2020)
28. Wolfram Research, Inc.: Mathematica. https://www.wolfram.com/mathematica/