A Framework to capture and reproduce the Absolute State of Jupyter Notebooks

Dimuthu Wannipurage
dwannipu@iu.edu
Cyberinfrastructure Integration Research Center, Pervasive Technology Institute, Indiana University
Bloomington, IN, USA

Suresh Marru
smarru@iu.edu
Cyberinfrastructure Integration Research Center, Pervasive Technology Institute, Indiana University
Bloomington, IN, USA

Marlon Pierce
marpierc@iu.edu
Cyberinfrastructure Integration Research Center, Pervasive Technology Institute, Indiana University
Bloomington, IN, USA

ABSTRACT

Jupyter Notebooks are an enormously popular tool for creating and narrating computational research projects. They also have enormous potential for creating reproducible scienti c research artifacts. Capturing the complete state of a notebook has additional bene ts; for instance, the notebook execution may be split between local and remote resources, where the latter may have more powerful processing capabilities or store large or access-limited data. There are several challenges for making notebooks fully reproducible when examined in detail. The notebook code must be replicated entirely, and the underlying Python runtime environments must be identical. More subtle problems arise in replicating referenced data, external library dependencies, and runtime variable states. This paper presents solutions to these problems using Juptyer's standard extension mechanisms to create an archivable system state for a running notebook. We show that the overhead for these additional mechanisms, which involve interacting with the underlying Linux kernel, does not introduce substantial execution time overheads, demonstrating the approach's feasibility.

CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**; **Open source model**; • **Applied computing** → **Enterprise data management**; • **Information systems** → **Data warehouses**.

KEYWORDS

Jupyter Notebooks, Apache Airavata, Reproducible Science

ACM Reference Format:
Dimuthu Wannipurage, Suresh Marru, and Marlon Pierce. 2022. A Framework to capture and reproduce the Absolute State of Jupyter Notebooks. In *Practice and Experience in Advanced Research Computing (PEARC '22), July 10–14, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3491418.3530296

PEARC '22, July 10–14, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9161-0/22/07…$15.00
https://doi.org/10.1145/3491418.3530296

1 INTRODUCTION

Project Jupyter's notebook format [14] has revolutionized interactive computing and has become ubiquitous among researchers using machine learning techniques and in scienti c computing communities. Jupyter's simple-to-use user interfaces, ease of deployment, rich visualization support, and support for multiple programming languages have nurtured a large user community with a diverse set of use cases, from simple Python code execution to complicated neural network simulations in high-performance computing environments. When code execution and visualization are combined with embedded textual descriptions, the Jupyter ecosystem for computing and data analysis can be viewed as an infrastructure for providing narration for the computing and data life cycle as stories [7, 19]. This storytelling is essential for reproducible science [17] or, even better or re-creatable, or tweakable science [2].

The high user adoption of Jupyter has resulted in a large ecosystem supporting multiple programming languages through kernel extensions and integration into users' development environments. Commercial cloud platform integrations such as Google Colaboratory [6] and a GitHub-based notebook repository MyBinder have fostered user sharing and collaboration of notebooks. Community support for popular data platforms has also enabled notebook reuse [13].

Reproducibility is an essential aspect in any science discipline and sharing someone's work through a notebook may be meaningful only if others can reproduce their work. While there has been a signi cant amount of work done in making Jupyter Notebooks con gurable and accessible to resonate with the user's requirements, there has been signi cantly less work to make notebooks reproducible across various platforms. We have identi ed several areas where reproducing a notebook execution is critical based on these considerations.

2 USE CASES FOR PORTABLE AND REPRODUCIBLE NOTEBOOKS

2.1 Reproducing Code from Scientific Publications

Software is a critical component of science, and reproducibility [10, 17, 18], and scholarly articles are increasingly referencing Jupyter Notebooks with code segments and visualizations. However, it is very cumbersome if not impossible to reproduce the published bundle of notebooks and associated code and data. Few challenges include recreating implicit data and identical runtime environments,

including a cascade of runtime library dependencies. Notebook kernels such as IPython rely on local Python virtual environments. Notebooks also interact with accessible external computational and data resources that are not easily translatable to shared collaborators. Out-of-band communication with authors of notebooks is necessary to fully reproduce the combination of data, dependencies, and runtime environment of the published work. Conducting these steps is a non-trivial and time-consuming process.

2.2 Sharing Classroom Learning Material

Jupyter Notebooks are a popular mechanism for providing classroom learning materials. Examples such as [3, 4] allow instructors to share notebooks, and students can execute them on their laptops during a classroom session. There are several cases when this process becomes signi cantly complicated. Students' laptops will have various operating system types and versions. Some may have di erent Python runtime versions, which may be incompatible with the instructor's notebook version. In some cases, instructors might need to use external data les as inputs for the code fragments in the notebook, and these also need to be shipped with the notebook code.

2.3 Transferring Local Computations to Between Local and High-Performance Computing Environments

Researchers have access to a scale of computing power from laptops to high-end supercomputers and computational clouds. With the profound usage of Jupyter, it is essential to support notebook portability across these resources without fragmenting data and code too much. As an illustrative use case, a researcher might start coding a notebook on a workstation, but training a higher-order neural network may necessitate a computer with a powerful GPU, large memory, and access to large data sets. Potentially, post-processing analysis may be bought back to a local laptop. In other cases, portions of the notebook may need to be run on a computer containing data that cannot be easily transferred to another place due to higher data volume or low network bandwidth. The data may need to remain under speci c security controls in some cases.

These mix and match executions within a single Jupyter Notebook are not supported in the standard distribution. Some work has been done in this area [4] to customize the vanilla iPython kernel and export the code execution to remotely hosted iPython kernel environments. However, these approaches require from the user a signi cant amount of e ort and expertise in distributed systems. In addition, these approaches focus on keeping the Python runtime context synced between external kernel environments but do not address any data-level dependency with the code fragments in the notebook. For example, when developing a neural network in a notebook, we use the training and testing datasets available in the local disk. Supposing others need to replicate the same environment, relevant input les must be manually selected. Then the notebook code must be updated with relative le paths and distributed with the notebook.

2.4 Running on On-Demand or Time-Bound Computing Infrastructure

Preemptable infrastructures like Amazon Spot Instances [1] or HPC shared resources with bounded maximum wall time do not guarantee the complete end-to-end execution of a long-running notebook. Users may also need to stop the notebook execution and create a snapshot to resume it from the last checkpoint at another time by reloading all the data and local context of the Jupyter session. In such scenarios, having a feature to create a comprehensive snapshot of the runtime saves time and computing resource utilization. When Colab Notebooks [6] expire after the threshold, the platform kills the notebook, and all the local states are lost unless the user manually serializes and uploads it to Google Drive. It would be bene cial to run the notebook from the last stopped checkpoint even after the original session has expired in all of these cases.

3 REPRODUCIBILITY OF NOTEBOOKS

Jupyter Notebooks inherently are sharable, but several missing components hinder the process of reproducing results in a di erent environment. Restarting a running notebook in a di erent environment can be made possible by ve essential requirements:

(1) The code within the notebook is portable.
(2) Python runtimes should be identical.
(3) Replicate referenced local data.
(4) External library dependencies should be replicated or installed if missing.
(5) Python runtime variables should be re-initialized.

Requirements 1 and 2 are external states which do not depend on the execution of the notebook. Requirements 3, 4, and 5 are internal to the Python runtime and the order of cell execution of the notebook (Figure 1). If any of the above ve requirements are not met, the replicated environment is not identical to the source environment. In most sharing solutions [23, 24] we can see that only requirements 1 and 2 are satis ed but not requirements 3, 4, and 5.

In this paper, we propose and implement a framework for Jupyter Notebooks to satisfy all ve requirements when replicating the state of a running notebook. In addition to the functional requirements, our solution is designed to prioritize the ease of use for users so that they do not have to put any more e ort into making this work than running a vanilla notebook environment.

4 A NOVEL APPROACH TO CAPTURE THE ABSOLUTE STATE OF NOTEBOOKS

To achieve the absolute state capture of a notebook, we primarily utilized the Jupyter Magic Extension support provided by the IPython kernel. In addition, we made modi cations to the iPython kernel to integrate with external services to capture information that cannot be directly captured from the kernel itself. We did not focus on steps 1 and 2 in the framework development as they are tightly coupled with the deployment. Current framework implementation, which is relevant to the scope of this paper, is available in the GitHub repository [25].

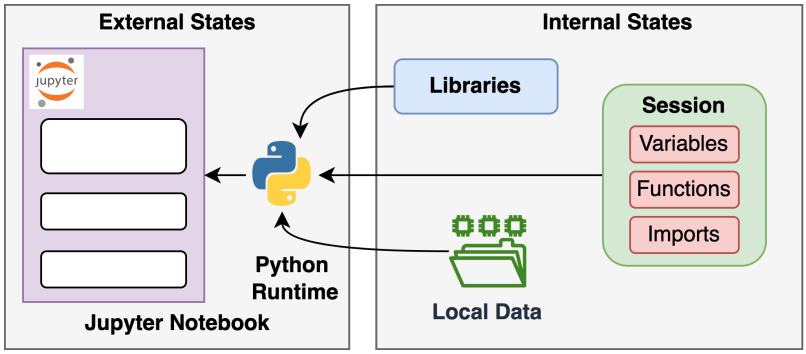

Figure 1: Components associated with the full state of a Jupyter Notebook. External states are not associated with the execution, and internal states depend on the code and execution order of the notebook.

4.1 Capturing Data Dependencies

Suppose a Python code in a Jupyter Notebook uses a le in the local le system for reading or writing purposes. In this case, this event should be captured if we want to replicate the data dependencies. However, this is a highly complicated task. There may be many Jupyter session-to- le system interactions either directly through the code the user has written or implicit libraries and function calls of the Python runtime. For example, a simple pip install command might access hundreds of les in the pip cache, and we should not consider them as data dependencies for the replication. Assuming that we have a way to lter out unwanted le system events, we still have the issue of capturing these le system events. Python and the iPython kernel do not have any native API to monitor le operations, so we must look for alternate ways.

4.1.1 Monitoring Linux Kernel System Calls by the IPython Kernel: All le operations are forwarded to the operating system as system calls regardless of the language-level le system operation implementation. To be precise, in Linux, all of these language bindings invoke the openat [15] system call to get the le descriptor of the le. We decided to capture the events from the Jupyter kernel process to the Linux kernel and log the invocations targeting openat system calls. We used Linux's strace [16] tool to lter and log the openat system calls coming through the IPython kernel process, which is responsible for the target Jupyter Notebook. There is a one-to-one relationship with a running Jupyter Notebook session and an IPython kernel process by design, so it is easy to derive le operations done by a notebook by capturing the procedures done by the designated IPython kernel process. Through this method, we managed to capture all the le system events coming from the Jupyter kernel runtime process irrespective of the language bindings used to access data at the programming language level.

The integration of the IPython kernel tracing framework is performed in two separate steps. In the rst step, we capture system calls sent by each IPython kernel and log them into individual process-speci c log les. In the second step, we analyze these logs les and identify relevant le system invocations.

Fig.2 illustrates the rst stage processes. We developed a process-tracing server, which can start Linux strace subprocesses for monitoring given process ids. These strace sub-processes capture the openat system calls and log them in a speci ed log le. When a user creates a Jupyter Notebook, the Jupyter Notebook Server initializes a dedicated IPython kernel process to handle requests. We updated the start method of the IPython kernel to invoke the process tracing server with its process id as a parameter. The communication between these components is performed through a Unix socket to minimize the communication overhead. Once the process tracing server receives the kernel start message, it starts a strace process to trace the IPython kernel process. Dashed arrows in Fig. 2 show the tracing connection created from the strace job. Once the events are captured, the strace job writes only the openat system call log to a designated log le. Dotted arrows in Fig.2 depict that connection. The strace Linux command needs to be executed as the superuser to access kernel system calls. For this reason, we separate the IPython kernel and process tracing servers into two separated entities; IPython kernels run as the regular user and the tracing service runs as the superuser.

The second major step analyzes these log les and captures relevant le system operations to create a list of les that the Jupyter Notebook accessed. To achieve this, we developed a Jupyter Magic [11] extension to parse the traced log le and lter out relevant le operations for further processing (Fig. 3). At any point in the notebook, users can run the Jupyter Magic line command to invoke this work ow and get a list of les that they directly or indirectly used in the notebook up to that point. We use a rule-based lter to ignore all operations not required to reproduce the notebook state. Discounted le operations include those associated with Python cache les, Jupyter Notebook checkpoint les, library les created and accessed as byproducts of notebook autosaves, and pip install commands for ltering relevant le operations.

These ltered les are used as the reference to create the export of data dependencies for the notebook execution, as explained in the next section of the paper.

4.2 Determining Dependencies on Optional Libraries

Python's runtime ships with a default set of libraries like math, io, and sys to perform common operations for mathematical, system, and le operations. In addition, many out-of-the-box libraries can be installed from the pipi.org repository [12]. Typically users use these

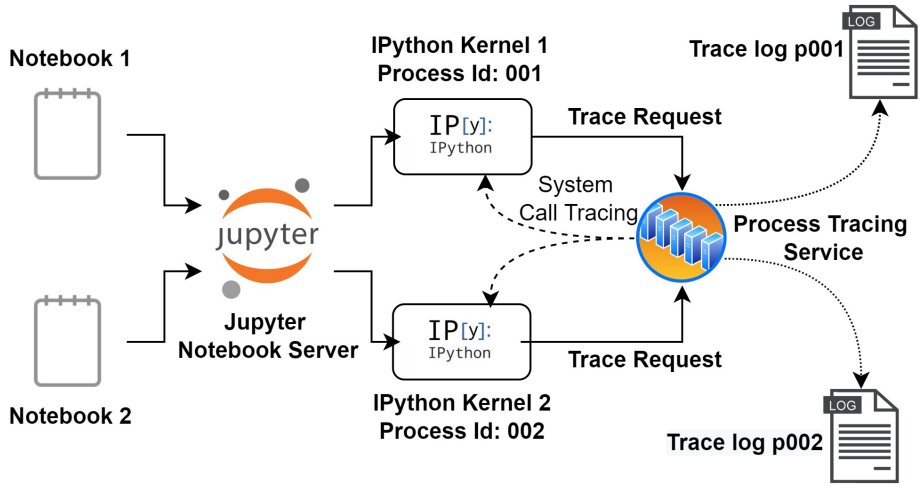

Figure 2: The workflow for capturing the filesystem calls from a Jupyter Notebook process to the operating system kernel. When an IPython process is started for a notebook, the process tracing service receives a notification to trace the process and then logs output into a file. Each notebook process has a different trace log file.

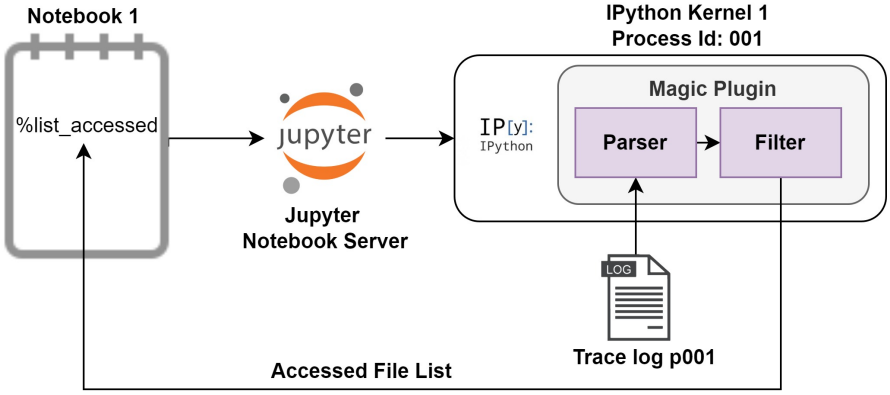

Figure 3: Workflow of processing a trace log file inside a Jupyter Magic plugin to shortlist the valid list of files accessed by the Jupyter Notebook. Trace log p001 is the log file associated with Notebook 1 that is created as depicted in Fig. 2.

libraries in notebooks either by installing them inside the notebook using the pip install command or using already installed libraries from the Python library cache of the running machine. However, each library may have multiple versions that contain some API-level and performance-related changes, so the code written using a particular library is somewhat coupled with the version of the library being used. When we share a notebook containing these library imports, there is no speci c way to guide the users on which libraries and versions need to be installed before running them. Typically, the approach is to run the shared notebook, and if there is a missing library in the Python environment, install the missing one using the pip install command. However, it is still not guaranteed to work properly as we do not know the exact version of the library that was installed in the author's environment. For this reason, syntax errors and performance issues are possible even though the relevant library is installed in the second system.

Considering the above concerns, deriving the optionally installed libraries and their versions loaded in a Jupyter notebook session is critical to reproducing the library dependencies. However, this needs to be carried out carefully. In Python, the library version loaded into the notebook's IPython session depends on the way it is being bootstrapped. If the notebook was installed in a Python virtual environment, dependencies are loaded from the virtual environment's context. If, instead, the notebook is executed in the user's default environment, libraries are loaded from Python's global library cache. The best way to derive the correct library version is to connect to the speci c notebook session and get the relevant library version.

To solve this problem, we developed another Python Magic extension that can be bundled into the running IPython kernel of the notebook. According to the IPython Magic framework, all the extensions receive the local namespace of the running IPython

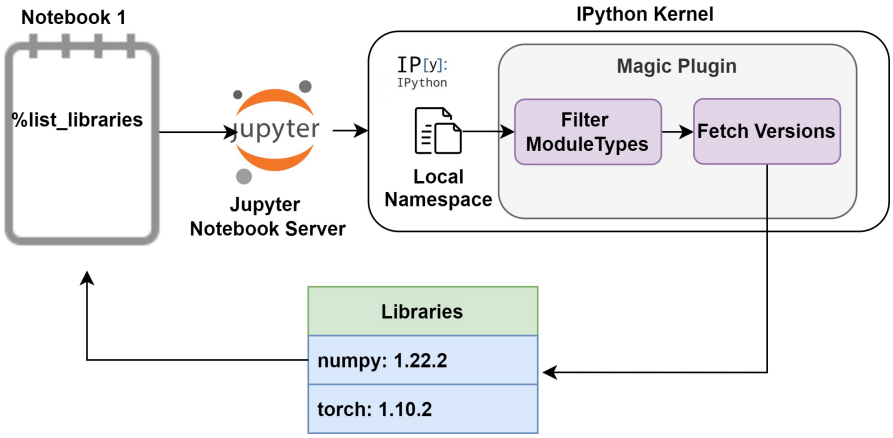

Figure 4: Magic plugin extensions capture the libraries' list and their versions imported inside a notebook session.

process. We can find all the imported libraries and versions inside this local namespace using a type-based filter below.

```
def imports():
    for name, val in local_ns.items():
        if isinstance(val, types.ModuleType):
            yield val.__name__
```

Next, we filtered out default libraries shipped with the Python runtime and captured the externally installed libraries. Once we get the library names, versions of the loaded library can be derived using the importlib [21] library.

Once all the relevant library names and versions are fetched, we save that metadata in a Python dictionary object where keys are the library name and values are versions (Fig. 4). This dictionary is used to export the library dependencies, as described in Section 5.

4.3 Capturing the Notebook's Runtime Session

The last and most challenging step to achieve complete reproducibility of a Jupyter Notebook is to capture the notebook's runtime session. This is an essential feature for checkpointing a notebook and restarting it in the same or different environment to continue from the last executed position. This requires serializing the execution session and reloading it back when it is restarted. A typical running Jupyter session may include library imports, initialized variables, and function declarations. If we want to create a snapshot of a session, we need a way to serialize all these types of entities when storing the state and deserialize them when restoring to a running session. In addition, we need to find a way to programmatically list the minimal set of entities that is sufficient to replicate the entire session.

A local namespace is a dictionary of objects where keys are entity names and values are entities created throughout the notebook session. These keys consist of variable names, function names, imports, and previous cells' execution ids. We developed a Jupyter Magic extension (Fig. 5) to capture this session information using the local namespace information that the extension receives from the Jupyter framework. Here we are interested only in the global variables, function definitions, and library imports required to recreate a session.

The challenge is to derive the subset of these entities from the local namespace as it represents all the entities as generic key-value pairs. To find this set of entity names, we use the `who` built-in Jupyter Magic command, which prints the signatures of imports, functions, and global variables. Once those entities are identified, we create a minimal version of the local namespace dictionary by filtering derived entity names. To export this sub-namespace object, we serialize it to a binary file using the `dill` [20] library, which is capable of serializing most of Python objects into binary format. However, there may be scenarios where some objects in the sub-namespace can not be serialized. To address this problem, we do a binary search on the sub-namespace and perform serialization tests on each subtree to determine the availability of any possibly incompatible entity. We remove any incompatible entities from the sub-namespace object and warn the user if there are any.

Once the sub-namespace object is serialized into byte format, we use it to build the notebook state bundle as mentioned in Section 5.

5 STRATEGY FOR REPRODUCING STATEFUL NOTEBOOKS

The previous section discussed how we capture the information to reproduce the whole notebook state. This section outlines how we package and export that information to be reproduced in a different environment. We also discuss how to bootstrap an environment when we import a notebook state from an exported bundle as mentioned above.

5.1 Packaging Strategy for Captured States

We use an archiving approach when exporting a notebook state that uses all of the above-mentioned state capturing techniques. First, all the files identified in the data dependency capturing stage are copied into the root level of the archive directory. To avoid conflicts, each file is duplicated with a unique UUID name rather than the original name. We also create a JSON file containing UUID-to-absolute-file-path mappings; this JSON file is also copied to the root level of the archive directory and the data. Library dependencies that we

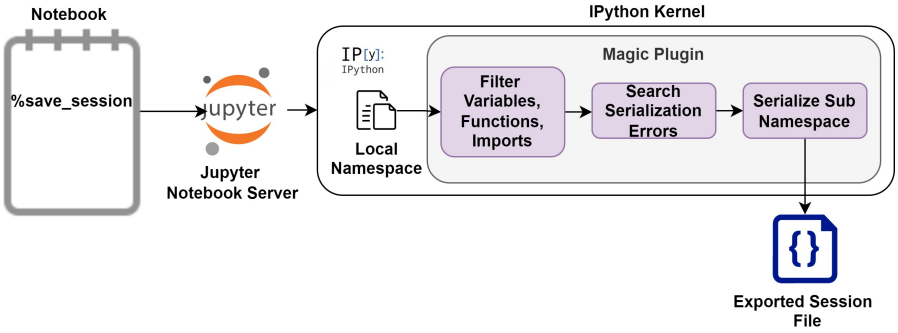

Figure 5: Magic Plugin implementation workflow to capture the variables, imports, and functions in the local namespace of the Jupyter Notebook session and serialize into a file.

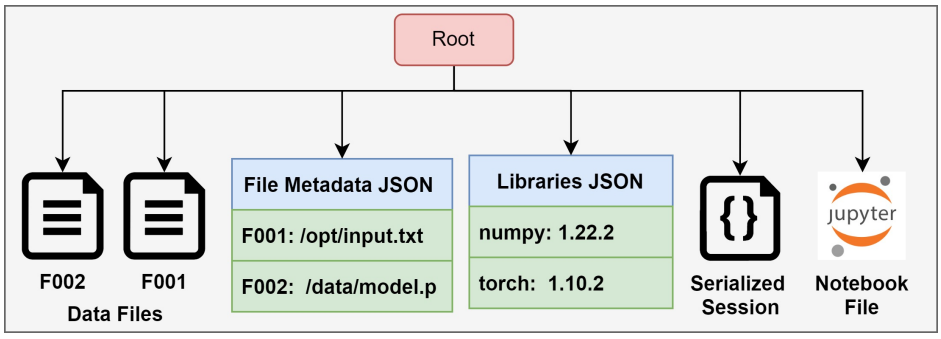

Figure 6: The structure of the archive file that contains a serialized stateful Jupyter Notebook

capture (as described in the previous section) are converted into a JSON format that contains library names and versions; this JSON file is also placed in the root level of the archive directory. In the notebook session capturing stage, we serialize the sub-namespace dictionary object to a file; it is also copied into the root level of the archive directory. Finally, we copied the notebook file as the last stage of archive creation. Figure 7 illustrates the contents and file structure of the archive.

To simplify the process, all of the operations mentioned above are integrated into a single Jupyter Magic command so that the user can invoke it inside the notebook file. Upon invocation, the custom Magic code internally invokes the state capturing magic commands described earlier, creates the archiving directory based on the data retrieved from those invocations, creates the archive file using the zip tool, and provides a link for the user to download it through the browser. This archived zip file can be later used to recreate the same state of the notebook in a different environment.

5.2 Recreating a Stateful Notebook from an Exported Archive

To recreate a notebook environment from an exported archive, we need to execute the following steps.

(1) Unarchive the exported archive file.
(2) Parse the File Metadata JSON file and copy data files into designated locations provided in the values of the JSON.
(3) Parse the Libraries JSON file and install those dependencies with versions using pip install commands.
(4) Start the IPython kernel and initialize a session to the exported notebook files.
(5) Run a Jupyter magic command to parse the serialized session and merge it with the local namespace of the notebook.

All of these steps can be performed in any notebook environment, including bare metal, cloud, and containerized platforms. We have developed a single script to accept the archive file with the notebook runtime to reproduce the archived environment. However, there may be issues when placing data files in Step 2 if the notebook is running in a different operating system that does not follow the file system structure of the source environment. For example, a notebook archive created on a Linux computer may fail at Step 2 on a computer running a Windows operating system.

Considering these scenarios, we recommend using a containerized environment when reproducing a notebook to make the initialization process independent of the host's file system. Both state-recording and state-replicating notebook environments can be bootstrapped from a one-line command to make the process as simple as possible for the user. To facilitate state recording and recreate a notebook, we have developed a container image that bundles a Jupyter runtime [5], a tracing server, and a python installation.

To start a notebook environment in state recording mode, users can run the command:

```
docker run --cap-add=SYS_PTRACE -it -p 8888:8888
    dimuthuupe/ipykernel:1.0
```

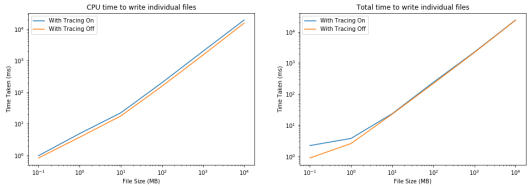

Figure 7: CPU time and total time to write a single file in various sizes using the notebook's Python runtime in tracing on and off scenarios

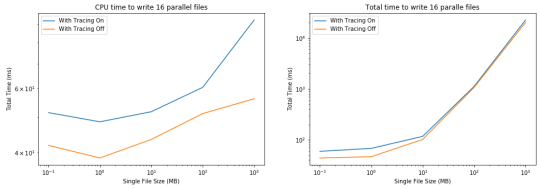

Figure 8: CPU time and total time to write 16 parallel files (1 per CPU core) in various sizes using the notebook's Python runtime in tracing on and off scenarios

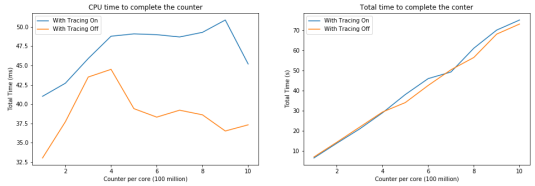

Figure 9: CPU time and total time run parallel counters (1 per CPU core) in various sizes using the notebook's Python runtime in tracing on and off scenarios

To recreate an environment from a state archive:

```
docker run --cap-add=SYS_PTRACE -it -p 8888:8888
-v <ARCHIVE_FILE>:/opt/ARCHIVE.zip dimuthuupe/ipykernel:1.0
```

6 PERFORMANCE ANALYSIS

We use an external Linux strace process to capture file operations performed by the notebook process to export the file list required to reproduce an identical environment. However, running an strace process along with the notebook to capture system calls may have performance implications because recording the operating system's system calls will cause them to block or wait. We record only openat system calls at the design level to mitigate the impact. However, there may still be a performance hit as everything from a mounted disk to a network socket in Linux is represented as a file. We may be capturing unwanted events that will eventually affect the notebook's overall performance.

We performed various stress tests on strace-enabled and disabled environments to evaluate this impact. We ran IO intensive, CPU intensive, and mixed workloads to identify areas affected by a bottleneck. All the performance tests were carried out on a workstation with 32 GB memory, 16 cores and disk write speed of 1GB/s. To keep both environments consistent, we ran both notebooks as Docker containers.

To enable and disable stracing, the container distribution of the kernel accepts the environment variable `ENABLE_TRACE` as a runtime configuration.

6.1 Running IO-intensive workloads

6.1.1 Single file write performance: In this experiment, we carried out a test to measure the performance impact from tracing system calls for a single file write. To simulate this scenario, we ran the following Python code fragment with various file sizes to measure the time it took to write a file into the disk with random characters. We captured CPU time and overall wall time for each run as metrics. CPU time is the time the code spends in the CPU, including the Linux kernel, and wall time is the total time it takes to run, including the CPU time and system call overheads (Fig. 7).

```
with open("random.dat", "wb") as output:
    output.write(np.random.bytes(FILE_SIZE))
```

6.1.2 Multiple file write performance: We then ran the same logic of writing files parallelly across 16 cores to measure the performance of parallel file writes (Fig. 8). This experiment fully utilizes CPU, memory, and disk IO, including multi-core executions, in-memory buffer creation, and parallel disk writes.

6.2 Running CPU Intensive workloads

To measure how the system performed when the CPU is fully utilized, we ran static counters on each CPU core parallelly using the following code fragment and measured the time taken for all the counters to finish (Fig. 9).

```
def f(x):
  counter = 1
  while True:
    counter = counter + 1
    if counter > COUNTER_SIZE:
      break
Pool(psutil.cpu_count()).map(f, range(processes))
```

6.3 Observations

We have analyzed three test scenarios: IO intensive, mixed (both IO and CPU), and CPU intensive operations to measure the impact of tracing the Jupyter notebook kernel process using the strace program to capture file operations. Among the three scenarios, we observed that IO intensive operations are the least affected by strace, while CPU intensive operations are the most affected in terms of the CPU time spent. However, measuring the total time to complete these operations, which is the real time between start and end, shows that significantly less cost is paid by the tracing enabled setup than a non-tracing environment. It is very clear that tracing affects CPU time, but because the proportion of CPU time to the total time is very low (the majority of total time is governed by the system call overhead), the impact of the CPU time does not significantly affect the final overall performance.

7 CONCLUSION & FUTURE WORK

This paper presents the framework-level concepts needed to implement a fully reproducible Jupyter Notebook environment and the results of a performance analysis of possible bottleneck scenarios. Currently, the proof of concept implementation of the framework supports Docker-based executions in single-user mode. Even with this limitation, our approach is potentially bene cial for long-running notebooks, especially on HPC resources: Users can pause running notebooks at some point and resume in a di erent environment with zero con guration changes. In the future, we plan to integrate this framework as a secure multi-tenanted service on the Jetstream2[8] cloud computing system with the scaling support of JupyterHub [12]. Integrating with Apache Airavata components, we plan to facilitate seamless remote execution of machine learning models on Jetstream2's GPU resources coupled with analysis on local resources all within a single notebook session.

REFERENCES

[1] Amazon. 2022. Amazon Spot Instances. https://aws.amazon.com/ec2/spot
[2] ReScience C. 2022. Reproducible science is good. replicated science is better. https://rescience.github.io/
[3] Julien Chastang, Rich Signell, and Jeremy Fischer. 2018. A unidata jupyterhub server: An online pyaos resource for students and educators. *98th American Meteorological Society Annual Meeting, AMS.*
[4] Renato LF Cunha, Lucas C Villa Real, Renan Souza, Bruno Silva, and Marco AS Netto. 2021. Context-aware Execution Migration Tool for Data Science Jupyter Notebooks on Hybrid Clouds. In *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 30 39.
[5] Dimuthu. 2022. Customized IPython Kernel Docker Image to capture Notebook state. https://hub.docker.com/r/dimuthuupe/ipykernel
[6] Google. 2022. Colaboratory. https://research.google.com/colaboratory/
[7] Brian Granger and Fernando Pérez. 2021. Jupyter: Thinking and storytelling with code and data. *Authorea Preprints* (2021).
[8] David Y Hancock, Jeremy Fischer, John Michael Lowe, Winona Snapp-Childs, Marlon Pierce, Suresh Marru, J Eric Coulter, Matthew Vaughn, Brian Beck, Nirav Merchant, et al. 2021. Jetstream2: Accelerating cloud computing via Jetstream. *Practice and Experience in Advanced Research Computing* (2021), 1 8.
[9] IPython. 2022. IPython "who" Magic Command. https://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-who
[10] Caroline Jay, Robert Haines, and Daniel S Katz. 2020. Software must be recognised as an important output of scholarly research. *arXiv preprint arXiv:2011.07571* (2020).
[11] Jupyter. 2022. Developing Custom Magic Extensions for Jupyter Notebooks. https://ipython.readthedocs.io/en/stable/con g/custommagics.html
[12] Jupyter. 2022. JupyterHub. https://jupyter.org/hub
[13] Jupyter. 2022. Kernels. https://github.com/jupyter/jupyter/wiki/Jupyter-kernels
[14] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* Vol. 2016.
[15] Linux. 2022. Openat System Call. https://linux.die.net/man/2/openat
[16] Linux. 2022. Strace - Process tracing command. https://man7.org/linux/man-pages/man1/strace.1.html
[17] National Academies of Sciences Engineering, Medicine, et al. 2019. Reproducibility and replicability in science. (2019).
[18] Roger D Peng. 2011. Reproducible research in computational science. *Science* 334, 6060 (2011), 1226 1227.
[19] Fernando Perez and Brian E Granger. 2015. Project Jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September* 11, 207 (2015), 108.
[20] Python. 2022. Dill: Python Object Serialization Library. https://pypi.org/project/dill
[21] Python. 2022. Importlib Library. https://docs.python.org/3/library/importlib.html
[22] Python. 2022. Pypi Repository. https://pypi.org
[23] David M. Rosenberg and Charles C. Horn. 2016. Neurophysiological analytics for all! Free open-source software tools for documenting, analyzing, visualizing, and sharing using electronic notebooks. (2016).
[24] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. , e1007007 pages.
[25] SciGaP. 2022. Framework to reproduce Jupyter Notebook Environments. https://github.com/SciGaP/iPython-Kerner-Changes

Checkpoint, Restore, and Live Migration for Science Platforms

Mario Juric, Steven Stetzler, and Colin T. Slater

DiRAC Institute and the Department of Astronomy, University of Washington, Seattle, WA, U.S.A; mjuric@astro.washington.edu

Abstract.

We demonstrate a fully functional implementation of (per-user) checkpoint, restore, and live migration capabilities for JupyterHub platforms. Checkpointing – the ability to freeze and suspend to disk the running state (contents of memory, registers, open files, etc.) of a set of processes – enables the system to snapshot a user's Jupyter session to permanent storage. The restore functionality brings a checkpointed session back to a running state, to continue where it left off at a later time and potentially on a different machine. Finally, live migration enables moving running Jupyter notebook servers between different machines, transparent to the analysis code and w/o disconnecting the user. Our implementation of these capabilities works at the system level, with few limitations, and typical checkpoint/restore times of O(10s) with a pathway to O(1s) live migrations. It opens a myriad of interesting use cases, especially for cloud-based deployments: from checkpointing idle sessions w/o interruption of the user's work (achieving cost reductions of 4x or more), execution on spot instances w. transparent migration on eviction (with additional cost reductions up to 3x), to automated migration of workloads to ideally suited instances (e.g. moving an analysis to a machine with more or less RAM or cores based on observed resource utilization). The capabilities we demonstrate can make science platforms fully elastic while retaining excellent user experience.

1. Introduction

With ever-increasing dataset sizes, remote analysis paradigms are becoming increasingly popular. In such systems (e.g. Jurić et al. 2017; Taghizadeh-Popp et al. 2020; Nikutta et al. 2020; Stetzler 2020), the users access data and computing resources through *science platforms* – rich gateways exposing server-side code editing, management, execution and result visualization capabilities – usually implemented as *notebooks* such as Jupyter (Kluyver et al. 2016), or Zeppelin. A challenge of this model is that the data provider (e.g., an archive facility) now bears both the cost of dataset storage and that of computing resources – including those used for running the users' Jupyter notebooks. This cost can balloon quickly, especially on cloud resources: left unmanaged, a 24/7 run for a 100 users reaches $\$300,000+$ range. This can be reduced by terminating inactive instances, but the price is a poor user experience.

In this contribution we present a solution: the ability to checkpoint (freeze) a user's running Jupyter notebook server to disk, and restore it to memory on-demand (including on a different host). This *C/R* functionality can dramatically reduce the cost, while fully maintaining the user experience. It also enables novel capabilities, such as uninterrupted migration of work based on resource needs.

2. Elsa: A Checkpoint-able JupyterHub Deployment

In a fully functional proof-of-concept we named *Elsa*[1], we added the C/R functionality to a cloud deployment of JupyterHub. The user experience can be viewed in a YouTube screencast at `https://dirac.us/5aj`; here, we provide a brief summary.

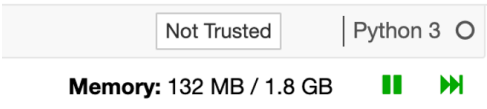

Figure 1. C/R UI. Pause checkpoints, fast-forward initiates migration to a different VM instance type.

With Elsa, the user logs into the JupyterHub aspect of the science platform and starts Jupyter on a machine with desired capabilities (e.g., CPU core count, or RAM size). The user then works on their notebooks as usual. However, an additional option is now present in the notebook interface – the "pause" button on the right of the notebook toolbar (Figure 1). Clicking this button checkpoints the complete state (memory, open files, etc.) of the notebook server, and releases all computational resources. Later, the user can restore the checkpointed session either on the same VM or one with different resources, and continue where they left off as if nothing happened.

3. Implementation

The high-level architecture of the Elsa prototype is shown in Figure 2. The JupyterHub front-end is run on a single, dedicated, node, from an essentially unmodified upstream JupyterHub container image.

Our main (configuration-level) customization is the addition of a new `Spawner` class. Within JupyterHub, a spawner is responsible for starting and managing users' notebook server instance(s)[2]. The spawner finds a VM where the notebooks run, and starts and stops Jupyter on that VM. As it has no support for pod C/R, we could not use Kubernetes at this time[3]; instead, our spawner directly allocates one new VM per user from the cloud provider. While reducing portability, running on bare VMs does bring some additional (and significant) benefits[4].

Although each user gets their own VM, per-user Jupyter is still run from a container. This i) abstracts away the details of the raw VM (e.g., Linux distribution doesn't matter, as long as podman/CRIU are available), ii) allows us to use the standard notebook-server container, iii) makes deployment significantly easier (a simple `pull`, rather than

[1]The source code is available at `https://github.com/dirac-institute/elsa`

[2]See `https://jupyterhub.readthedocs.io/en/stable/reference/spawners.html` for more.

[3]Work is ongoing; see `https://github.com/kubernetes/enhancements/pull/1990`

[4]Isolation between users at the VM level leads to predictable user experience; using VMs allows us to add swap making out-of-memory conditions a "soft" fail; and bare VMs are faster to provision relative to speeds from common `k8s` cluster autoscalers.

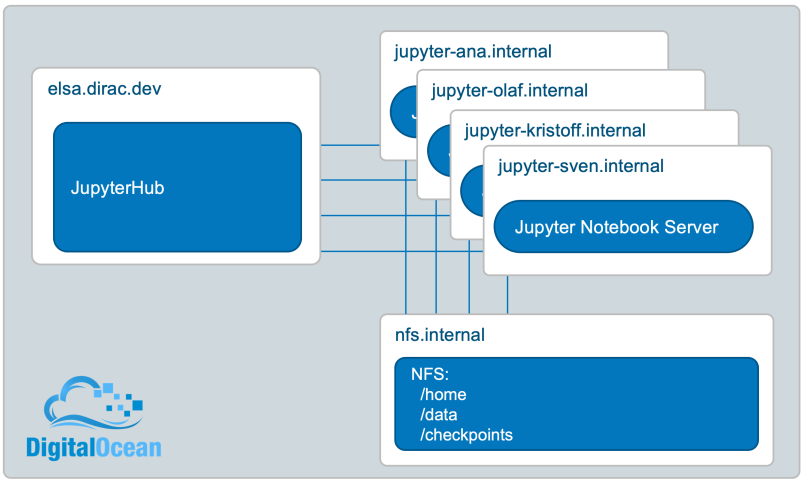

Figure 2. The architecture of the checkpointable JupyterHub deployment.

OS-level install), and iv) allows for secure re-use of VMs between different users (as users are sandboxed by their container).

We manage the container using `podman`[5], which has built in support for container checkpointing with `CRIU`[6]. To inject C/R support into Jupyter, we hijack the `Spawner's` start and stop APIs. When receiving the start command, our custom spawner restores the session from a previously stored checkpoint, if one exists. Similarly, upon receiving a request to stop it checkpoints rather than stops. This is clearly a convenient hack, and a proper C/R API should ultimately be added to JupyterHub.

Each VM mounts disks from a shared NSF server (`nsf.internal` in Figure 2), including the `/home` filesystem which is mounted as a volume within the user's container. A shared `/home` elegantly solves the problem of how to keep users' data identical on an inode level if/when they restore a checkpoint on a different machine (a requirement for checkpointing). While a shared filesystem introduces a potential bottleneck (e.g., imagine thousands of users simultaneously analyzing large datasets from their home-dirs), we haven't observed any issues in typical usage. For larger deployments, one could use a more scalable shared filesystem (e.g. `pNFS` or `GPFS`). We use the same NFS to centrally store the checkpoints themselves.

Finally, this new functionality is exposed to the user through a simple two-button UI featuring a "pause" and "fast forward" buttons (Figure 1). For simplicity, we added these to the `nbresuse` Jupyter extension.

This code has been prototyped and deployed on Digital Ocean[7] (`DO`). Given we use `DO` APIs in our spawner to manage the VM instances, Elsa will not run on other providers out-of-the-box. However, extensions to other clouds (e.g., AWS or GCP) are rather easy – O(50) lines of code – and planned as a future addition.

---

[5] A daemonless container engine for OCI Containers; `https://podman.io/`

[6] Checkpoint-and-Restore in Userspace; `http://criu.org`

[7] Digital Ocean is a low-cost cloud provider; `https://www.digitalocean.com/`

4. Discussion and Future Work

To our knowledge, this is the first fully functional implementation of checkpoint-restore and migration functionality of Jupyter notebooks on JupyterHub. It demonstrates that C/R for Jupyter not only possible, but fully functional for analyses as complex as the LSST software stack(Jurić et al. 2017). Going forward we plan to generalize the code to other Cloud providers, implement migration of open TCP network connections, add a dedicated C/R API for JupyterHub, and improve overall C/R performance.

We see three main application areas for this work: shared servers, on-prem science platforms, and Cloud-based science platforms. For shared servers (e.g., a machine used by a research group), one can now opportunistically checkpoint Jupyter instances after a period of inactivity thus optimizing overall resource usage. For on-prem science platforms our work lets the platform operator checkpoint rather than terminate inactive instances resulting in significantly better user experience. They can also dynamically migrate users' instances to optimize resource usage.

But the largest opportunity is for Cloud deployments, where C/R can both improve the user experience and significantly lower the operating cost. As we show in Table 1, with our C/R work running a typical user's Jupyter instance may cost as little as $200/yr, with no degradation to user experience relative to running 24/7. This is likely better than the total cost of ownership of running a similar system on premise, eliminating one more barrier to migrating science analyses to the Cloud.

Instance	Annual Cost
24x7x365 8core / 32G RAM (m5.2xlarge, on-demand)	$3364
As above + c/r, 15% duty cycle	$505
As above + c/r, 15% duty cycle, spot	$196
Savings	~ 17x

Table 1. Savings when running on cloud resources (AWS pricing as of 3:20pm PST, Nov 4, 2020.). The first row shows the cost of running an on-demand instance for an entire year (giving ideal user experience). The second row shows the cost of running for 15% of that time (a typical duty cycle we observed with our users), storing a checkpoint while the user is inactive. Finally we show the cost of running on spot instances, which is now possible as users work can be transparently migrated to a new instance if the spot VM is to be terminated.

References

Jurić, M., Dubois-Felsmann, G. P., D., Ciardi, & Guy, L. 2017, LSST Science Platform Vision Document. URL http://ls.st/lse-319

Jurić, M., Kantor, J., Lim, K. T., & et al. 2017, in ADASS XXV, edited by N. P. F. Lorente, K. Shortridge, & R. Wayth, vol. 512 of ASP Conference Series, 279. 1512.07914

Kluyver, T., Ragan-Kelley, B., Pérez, F., & et al. 2016, in PPAP, edited by F. Loizides, & B. Scmidt, 87. URL https://eprints.soton.ac.uk/403913/

Nikutta, R., Fitzpatrick, M., Scott, A., & Weaver, B. A. 2020, Astronomy and Computing, 33, 100411

Stetzler, S. 2020, A Scalable Cloud-Based Analysis Platform for Survey Astronomy. URL osf.io/e2zwf

Taghizadeh-Popp, M., Kim, J. W., Lemson, G., & et al. 2020, Astronomy and Computing, 33, 100412. 2001.08619

noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts

João Felipe Pimentel[1] Leonardo Murta [1] Vanessa Braganholo[1] Juliana Freire[2]
[1]Universidade Federal Fluminense [2]New York University
{jpimentel,leomurta,vanessa}@ic.uff.br juliana.freire@nyu.edu

ABSTRACT

We present noWorkflow, an open-source tool that systematically and transparently collects provenance from Python scripts, including data about the script execution and how the script evolves over time. During the demo, we will show how noWorkflow collects and manages provenance, as well as how it supports the analysis of computational experiments. We will also encourage attendees to use noWorkflow for their own scripts.

1. INTRODUCTION

Provenance helps users to interpret and reason about the results of computational processes [6]. For scientific experiments, it captures all computational steps and data that contribute to the output, thus enabling scientists to review these steps, input and intermediate data, and assess the quality of the derived results. For instance, if an experiment leads inconclusive results, the provenance data may help understanding the reasons. Moreover, when running several trials of a given experiment, scientists can cache intermediate data to avoid re-computing expensive operations [7]. Provenance also enables reproducibility by providing information regarding not only the computational steps but also the libraries and environment dependencies [3]. With such information, users can match the libraries and environment configurations to reduce external influences and reproduce experiments under similar conditions. After reproducing an experiment, provenance can be used to verify whether the same steps are executed, or if there are differences between the original run and the re-execution. Finally, scientists can use provenance to manage the evolution of experiments: they can create snapshots, restore specific versions of the experiment, libraries, and input data. In experiments that involve parameter exploration, scientists can create versions with each set of parameters, and use provenance to keep track of inputs and outputs [10].

Several approaches have been proposed to capture the provenance of computational processes. Tools that track provenance at the operating system level [8, 11] have two major benefits: they are general and language independent, and provenance can be automatically collected without requiring user intervention. However, the captured information can be hard to reason and to connect to the actual semantics of the experiments. Workflow management systems (WFMS) require scientists to specify their experiment as workflows [2, 19]. These tools support different levels of abstraction and when workflows are properly designed, the provenance closely matches experiment semantics. However, WFMS often require a steep learning curve and high adoption costs [17]. Furthermore, they lack the flexibility of general-purpose languages. For these reasons, many scientists still use scripts [5].

Tools have also been developed to track provenance from scripts [1, 9, 12, 18]. While some tools collect provenance automatically [12, 18], most focus on a single trial (i.e., a single execution of an experiment). As a result, these tools do not collect the information required to check for repeatability, do not support data re-use, and they are unable to manage experiment evolution. In addition to automatically collecting provenance from the execution of Python scripts, *noWorkflow* [12] addresses these limitations by tracking their history and evolution [14]. Thus, users can analyze multiple trials, compare them, and understand their history. In previous work, noWorkflow was extended to collect provenance and run analyses on interactive notebooks [16] and it was also combined to YesWorkflow [9] to link prospective provenance collected by YesWorkflow with retrospective provenance collected by noWorkflow [13].

In our demonstration, we will use real experiments as well as scripts provided by attendees and walk them through the process of provenance collection, analysis, and management supported by noWorkflow. We will show how provenance can be collected at different levels of granularity and showcase the different operations and visual representations noWorkflow provides to help users query and visualize provenance information.

2. OVERVIEW OF NOWORKFLOW

Collecting provenance of scripts is challenging. First, one must select the appropriate level of granularity. While coarse-grained provenance may hide important data, fine-grained provenance may overwhelm users. Moreover, scripts can encode control flow, cycles, and other structures that make it difficult to identify which parts of the scripts contributed to the generation of a given data product. Finally, scripts run outside controlled environments. Thus, it is hard to make assumptions based only on the results of scripts, as the environment may interfere with the execution.

Proceedings of the VLDB Endowment, Vol. 10, No. 12

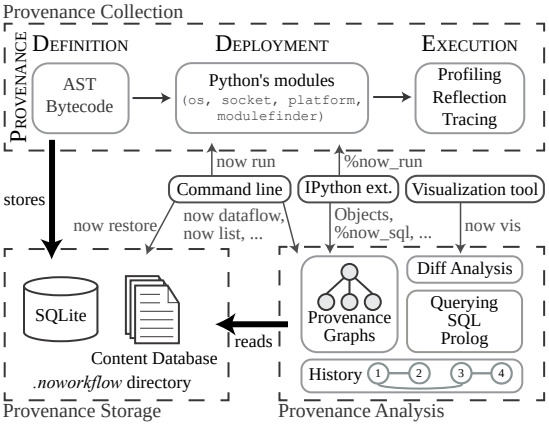

Figure 1: Architecture of noWorkflow.

noWorkflow addresses these challenges by collecting *definition*, *deployment*, and *execution* provenance. *Definition provenance* represents the structure of the script, including function definitions, their arguments, function calls, and other static data. *Deployment provenance* represents the execution environment, including information about the operating system, environment variables, and libraries on which the script depends. Finally, *execution provenance* represents the execution log for the script [12].

noWorkflow supports different techniques for collecting execution provenance at different levels of granularity. These techniques deal with the existence of control flows and cycles on scripts. Moreover, by collecting deployment provenance, noWorkflow is able to detect environment changes that are external to the script.

The architecture of noWorkflow has three key components, as shown in Figure 1. The *Provenance Collection* module collects provenance from scripts and stores it using the *Provenance Storage* module. The *Provenance Analysis* module reads data from the *Provenance Storage* and presents it to users in different ways. Users can interact with noWorkflow through three interfaces: command line, an IPython extension that interacts with Jupyter Notebook, and a web-based visualization tool. In addition to provenance collection and analysis, noWorkflow also collects the evolution history of experiments, allowing users to restore old files and to manage their execution.

3. DEMONSTRATION

In our demonstration scenario, a user receives a request from her collaborator to check if the precipitation in Rio de Janeiro remains constant across years. To verify this hypothesis, she collects data from a meteorological database and writes a script to process the data and produce a image for comparison. She starts the experiment with data from 2013 and 2014 and produces the script presented in Figure 2.

3.1 Provenance Collection

Running noWorkflow is as simple as running a Python script: instead of invoking `python experiment.py`, she invokes `now run experiment.py`. noWorkflow is able to run the very same Python script and produce the same results without modifications. However, instead of just running it, noWorkflow first generates a sequential trial identification number. Then, it collects the definition provenance and deployment provenance, and when it executes the script,

```
1  import numpy as numpy
2  from precipitation import read, sum_by_month
3  from precipitation import create_bargraph
4
5  months = np.arange(12) + 1
6  d13, d14 = read("p13.dat"), read("p14.dat")
7  prec13 = sum_by_month(d13, months)
8  prec14 = sum_by_month(d14, months)
9
10 create_bargraph("out.png", months,
11         ["2013", "2014"], prec13, prec14)
```

Figure 2: First script version.

it collects the execution provenance. As this is the first trial, the trial number will be 1. She can now use this number to reference the collected data for this trial. noWorkflow stores the collected provenance on disk in a directory named `.noworkflow` inside the script directory. This directory contains both a relational database for structured data and a content database for file-based data (see the Provenance Storage module of Figure 1). noWorkflow copies all accessed files, modules, and scripts to the content database during the execution of the scripts, and names them after the SHA1 hash codes of their content. These SHA1 hash codes link the content stored in the databases and avoid the duplicated storage of identical files.

noWorkflow collects the *definition provenance* by analyzing the Abstract Syntax Tree (AST) and the Python bytecode of the script [15]. It stores the script file and function definitions in the content database, and function names, calls, parameters, and global variables in the relational database. For this experiment, the content of 'experiment.py' is stored in the content database, while its hash code together with the function calls (i.e., `arange`, `read`, `sum_by_month`, and `create_bargraph`) are stored in the relational database.

noWorkflow collects two different types of deployment provenance: environment and library dependencies. As part of the environment, it collects operating system information (e.g., Ubuntu 16.04), hostname, information about the machine architecture (e.g., x86_64), Python version (e.g., 3.5.2), and environment variables. This data is stored in the relational database. noWorkflow also collects the transitive closure of all library dependencies together with their versions. In our example, it collects 'numpy' in version 1.11.3, 'precipitation.py' in version 1.1.0 (as it was declared by a __VERSION__ variable), and 'matplotlib' in version 2.0.0, among other internal modules. Note that 'matplotlib' was imported by 'precipitation.py'. noWorkflow stores the libraries in the content database, together with their names, versions, and hash codes in the relational database.

For execution provenance, noWorkflow stores copies of input and output files in the content database before and after reading or writing on them. Thus, it stores copies of 'p13.dat', 'p14.dat', and 'out.png' in the content database. In addition to the accessed files, noWorkflow supports collecting the execution flow at two different granularities. By default, noWorkflow collects provenance at a coarse granularity. It defines a Python Profiler that collects function activations (i.e., executed function calls), global variables, parameters, and return values. To collect provenance at a finer granularity, the user can issue the command `now run -e Tracker experiment.py`. Besides function activations, noWorkflow will also capture variable attributions, loop definitions, and other variable dependencies [15]. In this case, it combines the Profiler with a Tracer to store variable values

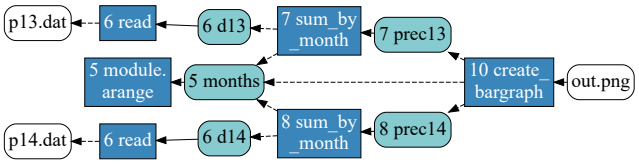

Figure 3: Trial 1 dataflow.

for each line and variable dependencies. A Tracer is commonly used for developing debuggers in Python. However, different from debuggers, noWorkflow stores the data values to assist users in future analyses. noWorkflow stores all data related to the execution flow in the relational database.

As a concrete example, consider line 5 of Figure 2. By default, noWorkflow captures that `np.arange(12)` returns a numpy array with values $[0, 1, \ldots, 10, 11]$. At fine-granularity mode, noWorkflow also collects the value of `months` as an array with values $[1, 2, \ldots, 11, 12]$. In both cases, noWorkflow collects the time of the execution and the function activation duration.

Provenance size may grow considerably if it is captured at a fine granularity or even at a coarse granularity, if script contains large loops. In order to avoid these problems, users can (i) limit the maximum collection depth of noWorkflow stack; (ii) write computational intensive functions in external files, since noWorkflow only collects execution provenance from main scripts, by default; or (iii) prepend an underscore to variables and function names, indicating which elements should not be collected.

3.2 Provenance Analysis

After collecting the provenance, noWorkflow offers multiple commands for provenance analysis. The user can run `now show 1` to obtain more details from trial 1 (textually). Users can also inspect individual modules, function definitions, environment variables, or function activations. This information can be visualized using the command `now dataflow 1 | dot -Tpng -o p1.png`, which produces a dataflow graph, as shown in Figure 3. In this figure, nodes with rounded corners represent data, white nodes represent files, light blue rectangles represent variables, and dark blue rectangles nodes represent function calls.

Since all provenance data, with the exception of file content, is stored in a relational database, the user can query the provenance using SQL. However, to enable complex, transitive closure queries, for which the support in some database systems is limited [4], noWorkflow also exports the trial provenance as Prolog facts using the command `now export 1`. Optionally, noWorkflow can export pre-defined Prolog rules that define common provenance queries together with the Prolog facts. For instance, the rule `access_influence(1, File, 'out.png')` indicates which files may have influenced the generation of 'out.png' in trial 1. In this case, the result is 'p13.dat' and 'p14.dat'. For obtaining a graphical representation of both SQL and Prolog schemas, users can use the command `now schema [sql,prolog]`.

After analyzing the experiment results, the user realizes that there was a drought in 2014 and decides to check if the precipitation remains constant when there is no drought. Thus, she adapts the script to include data from 2012. After running the experiment, she wants to compare the trials and present the differences to her collaborator.

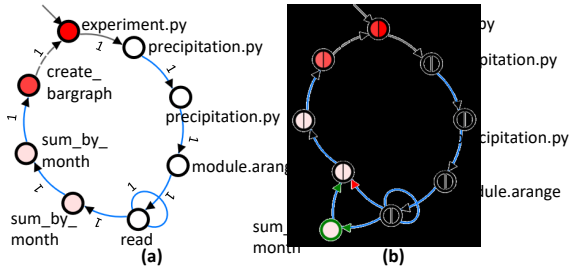

Figure 4: (a) Trial 1 activation graph and (b) Diff of trials 1 and 2.

For a textual trial comparison, she uses the command `now diff 1 2`. Currently, this command just compares basic trial information (e.g., parameters, duration), modules, and environment variables according to specified optional parameters.

However, she also wants to compare the execution provenance, so she uses the noWorkflow web visualization tool. This tool can be activated by the command `now vis` and further accessed on a web browser at http://localhost:5000. The tool presents the history of trials as a graph and allows users to select trial nodes to be visualized in more detail. When the user selects a trial, noWorkflow loads basic trial information, modules, environment variables, accessed files, and an activation graph. By selecting a second trial, the tool compares the first trial to the second one, presenting an activation graph diff [14] and all textual diff information. Different from the dataflow graph presented in Figure 3, activation graphs also work for coarse-grained trials. Figure 4 presents the (a) activation graph of trial 1 and its (b) diff to trial 2. In an activation graph, nodes represent activations and their colors represent their duration in a gradient scale: red represents the slowest activations, and white, the fastest ones. The script is an activation itself and it is indicated by a straight arrow. In this case, 'experiment.py' is the script. In the graph, black arrows represent the start of activations, blue arrows represent a sequence of calls within activations, and dashed arrows represent returns. Note that Figure 4(b) has an extra `sum_by_month` node and an extra read activation number in the loop edge.

3.3 Provenance Management

Now the user decides to change the script to add data from 2015. In the meantime, her collaborator realizes that there are unusual rainy days in the first trial and requests her to rerun the experiment without such days. Since she is using noWorkflow, she can restore the code and data from trial 1 by issuing the command `now restore 1` [14]. Since the user has changed the code, this command creates a backup trial, 3, with the modified script as definition provenance, before restoring the files from trial 1. By default, the command restores the whole trial to the state before its execution, but it supports optional arguments for restoring individual files, including intermediate and output files. Thus, the restore command is useful for trying alternatives on the experiment, for repeating trials, and for looking at old versions of trials. After restoring the trial, the user modifies the script and executes it again without the unusual rainy days.

noWorklow keeps track of the trial derivation history and allows users to visualize this history for understanding what happened to the experiment until it reached the current state. For visualizing the history, she can either run `now`

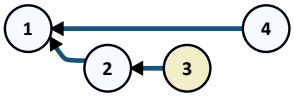

Figure 5: Experiment history with trials as nodes.

`history` and obtain a textual representation, or load the aforementioned visualization tool. Figure 5 presents the trial history for this demonstration. Note that trial 4 is based on trial 1 and trial 3 appears with a different color that denotes it is a backup trial. If the derivation history is not important, and the user just wants to list all trials with their command lines and durations, she can run `now list`.

Different from standard version control systems, noWorkflow versions are related to trial executions. This allows users to keep the full history of their experiments, keeping track of arguments, input data, output data, and other provenance information.

4. CONCLUSION

In this demonstration paper, we present *noWorkflow*, a tool that automatically collects provenance from Python scripts, without requiring any modification to the script. During the execution of scripts, noWorkflow collects imported modules, environment variables, function calls, file accesses, and, optionally, variables. While it does not collect network activity or database accesses directly, it collects the functions called for such accesses. noWorkflow also tracks the evolution of experiments and allows users to navigate over different versions. noWorkflow provides support for different kinds of provenance analyses through a command line interface, SQL and Prolog queries, and visualizations. Finally, noWorkflow also supports interactive analyses on Jupyter Notebooks.

noWorkflow is under active development. The system is available as open source software at `http://gems-uff.github.io/noworkflow`. Short videos showcasing the tool are available at `http://github.com/gems-uff/noworkflow/wiki/Videos`.

5. ACKNOWLEDGMENTS

We would like to thank CNPq, FAPERJ and the Moore-Sloan Data Science Environment for their financial support for this project. Juliana Freire is supported by the DARPA Memex and D3M programs, and NSF awards ACI- 1640864, CNS-1229185 and CNS-1405927.

6. REFERENCES

[1] C. Bochner, R. Gude, and A. Schreiber. A python library for provenance recording and querying. In *IPAW*, pages 229–240, 2008.

[2] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Managing the Evolution of Dataflows with VisTrails. In *ICDE*, pages 71–71, 2006.

[3] F. Chirigati, D. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *TaPP*, pages 977–980, 2013.

[4] S. Dar and R. Agrawal. Extending Sql with generalized transitive closure. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):799–812, 1993.

[5] S. Dey, K. Belhajjame, D. Koop, M. Raul, and B. Ludäscher. Linking prospective and retrospective provenance in scripts. In *TaPP*, pages 1–7, 2015.

[6] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

[7] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *ISSTA*, pages 287–297, 2011.

[8] P. J. Guo and M. Seltzer. BUrrIto: Wrapping Your Lab Notebook in Computational Infrastructure. In *TaPP*, volume 12, pages 1–7, 2012.

[9] T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, Y. Cao, F. Chirigati, S. Dey, J. Freire, et al. YesWorkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. *International Journal of Digital Curation*, 10(1):298–313, 2015.

[10] R. Meyer and K. Obermayer. pypet: A python Toolkit for Data Management of Parameter Explorations. *Frontiers in Neuroinformatics*, 10:1–16, 2016.

[11] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-Aware Storage Systems. In *USENIX ATC*, pages 43–56, 2006.

[12] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: capturing and analyzing provenance of scripts. In *IPAW*, pages 71–83, 2014.

[13] J. F. Pimentel, S. Dey, T. McPhillips, K. Belhajjame, D. Koop, L. Murta, V. Braganholo, and B. Ludäscher. Yin & Yang: demonstrating complementary provenance from noWorkflow & YesWorkflow. In *IPAW*, pages 161–165, 2016.

[14] J. F. Pimentel, J. Freire, V. Braganholo, and L. Murta. Tracking and analyzing the evolution of provenance from scripts. In *IPAW*, pages 16–28, 2016.

[15] J. F. Pimentel, J. Freire, L. Murta, and V. Braganholo. Fine-grained provenance collection over scripts through program slicing. In *IPAW*, pages 199–203, 2016.

[16] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire. Collecting and analyzing provenance on interactive notebooks: when IPython meets noWorkflow. In *TaPP*, pages 1–6, 2015.

[17] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with Vstrails. In *SIGMOD*, pages 1251–1254, 2008.

[18] M. Stamatogiannakis, P. Groth, and H. Bos. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *IPAW*, pages 155–167, 2014.

[19] M. Weske, G. Vossen, and C. B. Medeiros. *Scientific workflow management: WASA architecture and applications*. Citeseer, Universität Münster. Angewandte Mathematik und Informatik, 1996.

What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities

Souti Chattopadhyay[1], **Ishita Prasad**[2], **Austin Z. Henley**[3], **Anita Sarma**[1], **Titus Barik**[2]
Oregon State University[1], Microsoft[2], University of Tennessee-Knoxville[3]
{chattops, anita.sarma}@oregonstate.edu, {ishita.prasad, titus.barik}@microsoft.com, azh@utk.edu

ABSTRACT
Computational notebooks—such as Azure, Databricks, and Jupyter—are a popular, interactive paradigm for data scientists to author code, analyze data, and interleave visualizations, all within a single document. Nevertheless, as data scientists incorporate more of their activities into notebooks, they encounter unexpected difficulties, or pain points, that impact their productivity and disrupt their workflow. Through a systematic, mixed-methods study using semi-structured interviews ($n = 20$) and survey ($n = 156$) with data scientists, we catalog nine pain points when working with notebooks. Our findings suggest that data scientists face numerous pain points throughout the entire workflow—from setting up notebooks to deploying to production—across many notebook environments. Our data scientists report essential notebook requirements, such as supporting data exploration and visualization. The results of our study inform and inspire the design of computational notebooks.

Author Keywords
Computational notebooks; challenges; data science; interviews; pain points; survey

CCS Concepts
•**Human-centered computing** → **Interactive systems and tools; Empirical studies in HCI;** •**Software and its engineering** → *Development frameworks and environments;*

INTRODUCTION
Computational notebooks are an interactive paradigm for combining code, data, visualizations, and other artifacts, all within a single document [21, 36, 32, 30]. This interface, essentially, is organized as a collection of input and output *cells*. For example, a data scientist might write Python code in an input code cell, whose result renders a plot in an output cell. Although these cells are linearly arranged, they can be reorganized or executed in any order. The code executes in a *kernel*—the computational engine behind the notebook.

This interactive paradigm has made notebooks an appealing choice for data scientists, and this demand has sparked multiple open source and commercial implementations, including Azure,[1] Databricks,[2] Colab,[3] Jupyter,[4] and nteract.[5] While originally intended for exploring and constructing computational narratives [29, 31], data scientists are now increasingly orchestrating more of their activities within this paradigm [33]: through long-running statistical models, transforming data at scale, collaborating with others, and executing notebooks directly in production pipelines. But as data scientists try to do so, they encounter unexpected difficulties—pain points—from limitations in affordances and features in the notebooks, which impact their productivity and disrupt their workflow.

To investigate the pain points and needs of data scientists who work in computational notebooks, across multiple notebook environments, we conducted a systematic mixed-method study using field observations, semi-structured interviews, and a confirmation survey with data science practitioners. While prior work has studied specific facets of difficulties in notebooks [24, 17], such as versioning [18, 19] or cleaning unused code [13, 34], the central contribution of this paper is a taxonomy of validated pain points across data scientists' notebook activities.

Our findings identify that data scientists face considerable pain points through the entire analytics workflow—from setting up the notebook to deploying to production—across many notebook environments. While our participants reported workarounds, these were ad hoc, required manual interventions, and were prone to errors. Our data scientist report their key needs are support for deploying notebooks to production and scheduling time-consuming batch executions as well as under-the-hood software engineering support for managing code and history. Our findings further our understanding of requirements for supporting data scientists' day-to-day activities, and suggest design opportunities for researchers and toolsmiths to improve computational notebooks and streamline data science workflows.

STUDY DESIGN
Our investigation consisted of two studies. Study 1, a mix of complementary field observations and interviews, investigates the difficulties that data scientists face in their day-to-day activities. Study 2 confirms our findings from Study 1 through a survey of 156 data scientists.

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
DOI: https://dx.doi.org/10.1145/10.1145/3313831.3376729

[1] https://notebooks.azure.com/
[2] https://databricks.com/product/
[3] https://colab.research.google.com/
[4] https://jupyter.org/
[5] https://nteract.io/

Table 1: Field Study and Interview Participants

FIELD STUDY PARTICIPANTS					
ID	ROLE	INDUSTRY	EXP. (YRS)	NOTEBOOKS	LANGUAGES
FP1	Data Scientist	Advertising	5	Jupyter, RStudio	Python, R
FP2	Data Scientist	Cloud Computing	3	Jupyter, VS Code	Python
FP3	Data Scientist	Machine Learning	15	Jupyter	Python
FP4	Data Engineer	Machine Learning	3	Jupyter	Python
FP5	Data Engineer	Data Services	2	Jupyter, AzureML	Python
INTERVIEW PARTICIPANTS					
ID	ROLE	INDUSTRY	EXP. (YRS)	NOTEBOOKS	LANGUAGES
IP1	Cloud Soln. Architect	Cloud Computing	4	Jupyter, Zeppelin	Python
IP2	Data Scientist	Business Analytics	3	Jupyter, Databricks	Python
IP3	Data Scientist	Cloud Computing	4	Jupyter, Databricks	Python
IP4	Data Scientist	Security	10	Jupyter	Python
IP5	Data Scientist	Cloud Computing	4	Jupyter	Python
IP6	Soln. Architect	Development Tools	4	Jupyter, Colab	Python
IP7	Database Architect	Environmental Consulting	6	Jupyter	Python, Julia
IP8	Data Analyst	Entertainment	7	Jupyter, iPython	Python
IP9	Software Engineer	Manufacturing	8	Mupad	C#
IP10	Data Analyst	Finance	5	Proprietary	Python
IP11	Consultant	Finance	9	Jupyter, Databricks	Python
IP12	Consultant	Finance	15	Jupyter	Python
IP13	Data Scientist	Security	10	Jupyter, Colab	Python, R
IP14	Software Engineer	Cloud Computing	9	Databricks	Python
IP15	Data Scientist	Development Tools	5	Jupyter, Databricks	Python, R

Study 1: Field Observations and Exploratory Interviews

To understand when and why data scientists experience difficulties with notebooks, we conducted field observations to observe data scientists in situ. We complemented these observations with interviews with professional data scientists to get a broader picture.

Recruitment. For our field observations, we recruited five professional data scientists from within Microsoft—a large, multinational, data-driven organization—using our internal address book to sample data scientists with the title of "Senior" (or higher) and having at least two years of experience in the organization. For our interviews, we recruited 15 data scientists having at least two years of data science experience from multiple companies and industries.

Participants. Participants reported working in a variety of industries and roles, including Data Analyst, Data Scientist, and Data Engineer (Table 1). Participants reported 6.6 years of experience ($sd = 3.7$). Participants primarily reported using Python with Jupyter notebooks.

Field study protocol. We observed data scientists who primarily work in computational notebooks as they performed their regular data science activities. All sessions were conducted with a single observer and a single data scientist in the data scientist's office. Sessions were scheduled for one hour, with 45 minutes of observation and 15 minutes of retrospective interviews. During the session, we recorded their screens and audio through screen capture software and a hand-held audio recorder. We asked participants to think-aloud as they worked, and the observer took in-situ field notes as the data scientists conducted their work. During the retrospective interview, the observer used these notes to probe further about difficulties observed or mentioned during the session.

Interview study protocol. We conducted semi-structured interviews remotely through online communication software, and recorded these interviews. The questions roughly followed this organization: brief questions regarding what they do as a data scientist and what activities they conduct using notebooks, followed by more detailed conversation about why they prefer to use notebooks as well as any difficulties when using the notebooks. Interviews were about 30-45 minutes long.

Informed consent. In both field observations and interviews, participants signed a consent form prior to conducting the study, in accordance with our institutional ethics board. Participation in the study was voluntary and participants received no compensation.

Analysis. We transcribed the audio for the field observations and interviews. The first and last author collaboratively analyzed these transcripts through an inductive, open coding process using the ATLAS.ti qualitative analysis software. First, we segmented the transcripts and applied descriptive codes [35], that is, assigning short phrases or labels, to these segments. We merged and split descriptive codes as necessary, looking for similarity in challenging *activities* that data scientists experienced when working in the notebook. Next, we performed axial coding, grouping similar codes and analyzing them to identify higher-level and cross-cutting themes,

which we termed *pain points*. The collaborators met frequently over several weeks to discuss, examine, and refine codes and themes.

Validity of qualitative coding. To support interpretive validity, we recruited two external raters, both data scientists. We randomly sampled five statements from each of the nine pain points as identified in our inductive coding process. Using Table 2, we asked the raters to independently categorize each statement and assign it to a single pain point that *best* reflected that statement. We achieved a Cohen's Kappa just below 0.8, with disagreement from our external raters primarily because some statements concern multiple pain points.

Study 2: Confirmatory Surveys

To triangulate, validate, and increase the credibility of our qualitative field observations and interviews, we conducted a survey with a broader population of data scientists.

Survey protocol. Our survey consisted of demographic questions about the respondents' number of years of data science experience, the computational notebooks they use regularly, the programming languages they use regularly within these notebooks (multi-option question, with "other" as an answer choice), and how frequently they use computational notebooks for their data science activities (5-point Likert-type item frequency scale, "More than 5 times a week," "4-5 times a week," "2-3 times a week," "once a week," and "less than once a week."

Using the findings of our qualitative analysis of field observations and interviews, 20 activities were drawn and presented as a series of questions using a 5-point Likert-type scale for difficulty and importance. All questions in the survey were optional.

To evaluate if our field observations and interviews reached theoretical saturation, we asked respondents to indicate if there were any other difficulties with using computational notebooks that we missed.

Informed consent. Our survey included instructions for informed consent, in accordance with our institutional policies.

Recruitment. We recruited participants by e-mail through internal address book contacts across multiple organizations, through social media such as Twitter and LinkedIn, and through data science mailing lists. We also asked respondents to forward the survey to other data scientists. Respondents did not receive compensation for completing the survey.

Respondents. 156 data scientists from various companies responded to our survey, after discarding eight blank responses. Respondents had an average of 5.3 years of experience ($sd = 3.9$). 53% of our respondents used notebooks "more than 5 times a week" while 16.6% of them used notebooks once or less than once a week. 98% of our respondents primarily used Python in notebooks, 30% used R, and 14.7% used Scala. Respondents also reported using Java, JavaScript, Spark, and SQL. 84% used Jupyter notebooks and 33% used Jupyter Labs. 36% reported using Databricks and 28% use Azure Notebooks.

Analysis. We computed descriptive statistics and plotted the survey responses for difficulty and importance. We manually inspected the responses for other challenging activities that we missed. Respondents rarely populated this response. In all cases, the answers were additional details for already-covered activities ("restructuring the code" for the activity of "refactoring" and "using loops to create multiple plots" for "visualizing data and models") or confirmational responses ("looks like you've covered everything").

PAIN POINTS OF USING NOTEBOOKS

We identified nine categories of pain points in computational notebooks, across the data scientists' workflow (Table 2).

Setup

Difficulties with notebooks happen as soon as the data scientist creates a new notebook.

Loading data. To explore data, it first has to be pulled into the notebook. That's not always easy, especially when the data needs to be shuffled back and forth between multiple sources and platforms (IP10, IP11, IP14, IP15, FP1). This process quickly becomes a tortuous, multi-step adventure that requires repeatedly "going to separate cloud instances to bring down the data locally, taking that to a local file, and uploading it back to the cloud" (IP15). Although some data libraries exist (for example, `psycopg2`)[6], they are nontrivial to use, and data scientists must be aware that they exist and remember how to use them. Unsurprisingly, some of our data scientists relied on others for help—IP10 had a developer build *magic commands* in the notebook that "triggered functions behind the notebook" on their behalf and provided them with an easier-to-use interface to connect to their commonly-used data sources.

Sometimes, when working with large data sets, the "notebook tends to crash a lot; the kernel dies and that causes frustration" (IP13). In contrast to IDEs, the client/server nature of notebooks complicates setup, and the difficulties of working with large data within a web browser are amplified. In such cases, this often "requires getting a lot of data engineering resources just to be able to run something that's supposed to be a daily job. The stack to do something pretty simple is pretty heavy" (IP15). But not all data scientists have dedicated developers or data engineers to help them.

Cleaning data. While clean data makes "a big difference in the overall model output" (IP15), it's seldom readily available. Efforts to clean data are mostly clerical, and there's "no mystery—it's just time consuming" (IP1). To avoid repetitive cleaning data scientists create "a bunch of routines" (IP11). But these routines still require modification and manual copying-and-pasting across notebooks (IP4, IP9, IP15)—an error-prone process.

Explore and Analyze

Although notebooks profess to allow "quick and dirty work and exploration" (IP1, IP3, IP4, IP6, IP8, IP11 , IP12, IP13, IP14), data scientists tell us that this isn't always the case.

[6] http://initd.org/psycopg/

Table 2: Summary of Pain Points in Computational Notebooks

PAIN POINT	DESCRIPTION	EXAMPLE
Setup	Loading and cleaning data from multiple sources and platforms is a tortuous, multi-step, manual process.	"If you do a lot of data loading and pre-processing always re-loading the data is time consuming" (IP2).
Explore and Analyze	An unending cycle of copy-paste and tweaking bits of code made worse by feedback latency and kernel crashes.	"I need immediate feedback, like when I am testing slight changes in the model. I don't want to execute everything again" (IP1).
Manage Code	Managing code without software engineering support results in "dependency hell" with ad hoc workarounds that only go so far.	"Debugging is a horrible experience, copying the code over to do the debugging outside [in the IDE], and copying it back" (IP8).
Reliability	Scaling to large datasets is unsupported, causing kernel crashes and inconsistent data.	"Disconnects between browser-server or server-kernel introduce all sorts of lack-of-reliability problems" (IP6).
Archival	Preserving the history of changes and states *within* and *between* notebooks is unsupported, leading to unnecessary rework.	"The thing is using any kind of versioning mechanism for notebooks is just a complete and utter failure" (IP2).
Security	Maintaining data confidentiality and access control is an ad hoc, manual process where errors can leak private client data.	"We are missing a more private way of handling credentials. I don't want client credentials be visible to others" (IP13).
Share and Collaborate	Sharing data or parts of the notebook interactively and at different levels—demo/reports, review/comment, collaborative editing—is generally unsupported.	"There are cases where somebody is asking you to review/comment, while other times to go collaborate" (IP6).
Reproduce and Reuse	Replicating results or reusing parts of code is infeasible because of high levels of customization and environment dependencies.	"The fact that somebody could run a notebook on organization A's service but not on organization B's is a serious problem" (IP6).
Notebooks as Products	Deploying to production requires significant cleanup and packaging of libraries—DevOps skills that are outside the core skill set of data scientists.	"Once the code gets a certain level of maturity, it's very difficult to transition that to production code. Everything has to translate to functions and classes" (IP15).

Modeling. Building models take time. Not only is "[having the system] learn the model itself very time consuming" (IP7), it also "involves a lot of complexity" just to build them (IP3). Getting to the right models require many iterations, but data scientists don't get "immediate feedback" so that they can quickly make adjustments (IP1, IP2). Instead, data scientists like IP1 have to wait a long time to check if the execution was successful, and they can't interrupt the process to evaluate alternatives in the meantime. Worse, if their model produces an error, they have to "execute everything again" (IP1).

Visualizing. Data scientists use visualizations—primarily plots—to quickly see how their code refinements modify their data (IP1, IP7, IP8, IP6, IP12, IP13). But it's hard to customize the plots when the data scientist isn't happy with the result; these frustrations led our data scientists to continuously copy-paste or tweak bits of code (IP4, IP12, IP15) to tailor the visualizations to their needs. In some cases, the output cells themselves are a hindrance—"there's a lot of things just limited by the footprint of the notebook. Everything is in a cell, and the chart is limited by the boundaries and real estate of the notebook" (IP15). In these situations, data scientists end up manually exporting their code and data and redoing the work in exploratory data analysis tools outside of the notebook—"integration with tools like Tableau, or even API level access to them, would reduce all this copying and pasting" (IP8). Since notebooks are intended for facilitating data exploration, it is unfortunate that visualization continues to be difficult within some notebook environments.

Iterating. Having to iterate between modeling and visualization, that is, "changing some methods slightly and trying different things" (IP1), is the norm. Of course, supporting iteration is one of the core purposes of computational notebooks. Notebooks should be an ideal environment for iterating, but like we saw in setup, churning through code introduces many of the same difficulties when code assistance isn't available: the data scientist ends up "having to go through the same ceremony to do even the most basic modeling task" (IP8), finding relevant packages, and deleting now-unused code (IP2, IP8, IP9, IP11, FP3). One option is to switch to an IDE, but

this requires constantly shuffling between the IDE and the notebook.

Manage Code

Although managing and working with code is a fundamental activity in the computational notebook paradigm, data scientists told us about code-related activities they found to be challenging.

Writing code. Having to write code—particularly due to lack of code assistance—is something that IP7 "hated the most" about working in notebooks. To be efficient, they had "to know all the function names and class names correctly and have another browser open to search for help and documentation" (IP7, FP2). Coding in notebooks is even more difficult using new libraries since it's not possible "to explore the API and functions" from within the notebook (IP8). Practically, IP8 argues, "anyone who tries to use notebooks has to start off with an IDE and then graduate into a notebook."

Managing dependencies. Having to manage packages and library dependencies within the notebook is, to put it mildly, a "dependency hell" (IP7). Notebooks provide little-to-no-support for finding, removing, updating, or identifying deprecated packages (IP3, IP7, IP9, IP11, IP12, IP13, IP15). Often, discovering what packages are even installed isn't accessible from the notebook environment, requiring data scientists to plod over to their command-line terminal and use commands like `conda` and `pip` to manage their environment (IP3).

Debugging. Feedback about debugging in notebook was mixed. Some data scientists applauded the notebook's ability to quickly diagnose errors (IP2, IP5, IP14), while others maintained that it's "a horrible experience" (IP6, IP8, IP15).

On one hand, the cell-oriented structures make debugging "fairly instantaneous and straightforward" because it allows "splitting up functions into different cells and then slowly stitching them together until you get something that works right" (IP2). In the case of errors, notebooks at least usually retain the output states of previously executed cells. On the other hand, the only way to debug in most notebooks is through the use of `print` statements—many computational notebook don't let data scientists "peek inside variables and change them" (IP7). Due to out-of-order execution, typical IDE affordances like breakpoints would make it difficult to "follow the code flow" (IP2), and notebooks likely require different affordances to support debugging in this exploratory context (for example, data introspection).

Testing. Tests are another mechanism to troubleshoot issues in notebooks, but because there's no standard way to test notebooks, different data scientists end up following different approaches (IP1, IP3, IP4, IP5, IP6, IP9, IP13, FP1). For example, while IP13 and FP1 wrote test cases within the same notebook, IP3 and IP4 used dedicated test notebooks to validate their functionality.

Reliability

One problem with computational notebooks is that executing them "isn't particularly reliable" (IP6). The other is that they don't scale to big data.

Executing notebooks. If the notebook kernel crashes during the middle of an operation, or the data scientists gets disconnected—this can result in putting the notebook or the data in an inconsistent state. For example, when inserting many records into a database, getting disconnected from the notebook can result in only partial records being written to a data store (IP10, FP4). Inconsistent state can be hard to detect because there's "no transparency in terms of understanding how the process is being executed on the kernel" (IP2). And some notebooks "get very large due to people abusing them; as notebooks get larger, the reliability falls" (IP6). Sometimes it's easiest to just restart and run the whole notebook again (IP8, IP10, IP15).

Scaling to big data. A limitation to iterating is that notebooks "can only handle so much data in the notebooks" (IP8, IP10). "Although notebooks could be used for lightweight extracting, transforming, and loading data (ETL), for heavyweight ETL we still have to rely on the Java pipelines. The data is way too huge for notebooks to handle" (IP6). Notebooks introduce a tension between balancing the needs of quick iteration and working with large data (IP4).

A key reason was that reliable kernel connections were hard to maintain and resulted in kernel crashes (IP1, IP2, IP6, IP10, IP13). These crashes were often a result of the notebooks' limited processing power, which can't handle large notebooks or big data loads.

Archival

While some exploratory notebooks have a relatively short shelf life as "playgrounds" (IP5), other notebooks have longer lifetimes. For the latter, data scientists need support for versioning and searching notebooks.

Versioning. There's "a lot of room for improvement when we want to check notebooks into source control, such as being able to visualize the differences between the last version and the new version" (IP3). Using traditional versioning mechanisms intended for source code are "just a complete and utter failure" (IP2) when versioning notebooks. IP2 continues, "because all the outputs are saved within the notebook, there's a lot of state that's bundled in the file." In traditional source control systems, all of these changes appear as spurious differences, making it difficult to identify the actual changes between the notebooks—"there's just a a lot of mess" (IP2, FP2). To be effective, version control systems need special-handling for computational notebooks. Moreover, since "the history and execution order of the notebook cannot be tracked by version control systems" (IP2, IP15), committing the notebook to version control doesn't mean that you'll be able to successfully run the notebook.

Searching. Data scientists create a lot of notebooks, and these notebooks are "rarely maintained or given useful names, which make it hard to know what's saved in these notebooks" (IP9). Since the "number of notebooks grows quickly" (IP7), folders and files names "become disorganized very fast. It's hard to remember what is saved in these things, so they're all just `Untitled-1` or `Untitled-2`" (IP9). All of which make "finding and navigating to the intended file difficult" (IP14).

Security

Securing notebook was difficult along two dimensions: avoiding unintentional data leaks and managing access to the notebooks.

Securing sensitive data. Securing data required our data scientists to perform "additional suppression and transformation operations," which is both time consuming and computationally costly (IP11). Adding security into the mix "slows down the whole [data science] process" (IP11). In some cases, it's the transformation code itself that needs to masked as it can reveal patterns of the underlying suppression mechanisms. But notebooks don't provide a way to "lock the functionality of a particular cell so that it's not visible to others," while still allowing them to execute the code (IP13). For certain types of analysis, the transformed data had to be again "unmasked," which introduced even more steps into the data science process. Finally, copying or exporting desensitized data over to external platforms can be "dangerous, because if you're copying data and screw up something," this data can be inadvertently leaked (IP8).

Controlling access. In some notebook environments, it's not possible to finely control access to the notebooks, and "the only way to share notebooks securely is to upload notebooks to secure team drives" (IP6, IP10, IP11, FP5). For example, data scientists aren't able to specify the types of activities that others can perform within a notebook, such as making the notebook "read-only or only allowing comments" (IP12) or restricting notebook to "only run but not modify cells," or modify cells but not run (IP10, IP13).

Share and Collaborate

Sharing and collaboration is "messy" (IP5) and "not straightforward" (IP2) but "very often requested" (IP6).

Sharing supporting artifacts. "Sharing a notebook is kind of useless without the underlying data" (IP9). But doing so means that the recipient has to be able to have access to the database, and the correct data within it (IP9, IP15). In addition, data scientists also need to "match the environment settings" of the notebook (IP3, IP15), for example, by following manually-written instructions. Alternatively, the sender can create separate "requirements files" which can be read by package installation programs to automatically install the necessary dependencies (IP15). Unfortunately, this approach breaks the "single file" metaphor of notebooks, because information needed to run the notebook is decoupled from the notebook itself.

Sharing with non-data scientists. Data scientists need to present their notebooks to customers and other non-data scientists. They do so in three ways: interactively (where the data scientists execute the notebook directly to explain the results to the audience), statically (for the audience to look at the analysis and results at a later time/offline), or as a read-only notebook (for non-data scientists to leave comments or notes on the notebook for the data scientist to review at a later time).

When interactively sharing notebooks through live demos, data scientists desired to execute the notebook interactively without exposing code, since the code isn't meaningful for most of the audience. Data scientists wanted the ability to temporarily "collapse code and show only the visualizations and markdown" (IP3, IP7, IP15).

Data scientists also shared their results by exporting the content of notebooks to other formats, such as presentations or printable reports. But because visualizations and data in notebooks "spit out a lot of information, it's hard to create nice looking reports" (IP4).

Finally, some data scientists wanted stakeholders to be able to leave comments on the notebooks without being able to run the cells. "Giving the non-technical users an opportunity to provide feedback on the notebook itself by leaving comments can be extremely valuable" (IP3, IP6).

Editing collaboratively. "Collaborating in real-time with multiple users on a single notebook is super powerful" (IP6). However, collaboration is not possible in many notebook environments, which makes it challenging to "develop side-by-side with somebody, especially remotely" (IP3).

Reproduce and Reuse

Data scientists reproduce notebooks to verify how a result was computed or obtained. They also reuse and adapt code from existing notebooks to save time.

Reproducing. "Reproducibility is really ensuring that the set of software you used to produce a result is exactly the same the next time you try to reproduce the same result" (IP6). "The only way another person can run the notebook is if they're able to match all the environment settings" (IP15). Another challenge to reproducibility is that data scientists customize notebooks, for example, by using extensions that they install on their local notebook environment. Depending on the extension, these notebooks can fail to run if they are not available in the other notebook environment (IP2, FP5). Crucially, "there's no compatibility at all between extensions in different notebook implementations. There's absolutely no standard for how these extensions work. So chances are it's not gonna work. That's a serious problem" (IP6).

Reusing and adapting. Data scientists reuse notebooks, for example, in situations where "the data source can change, but the underlying logic would be the same" (IP11). Unfortunately, even seemingly simple reuse can become more complicated than expected, such as when the "earlier notebook uses absolute paths" (IP5), when the "cells have no designated format or function" (IP1) and can't be easily isolated, and when there are complex dependencies to bring into the new notebook (IP3, IP11, FP4).

Notebooks as Products

Notebooks shine when performing quick, interactive, exploratory analysis, but these features also encourage sloppy coding practices that make it difficult to transition notebooks to production.

There are two common pathways to production. First, the data scientist can run the notebook directly, but with larger data and with longer-running computations. Second, they can package the code in the notebook into a standalone artifact

that is decoupled from the original notebook. Neither option is particularly pleasant.

Scheduling long-running computations through the notebook. Running long-running computations in notebooks provides no feedback on progress; while the computation is running, data scientists lose all interactivity in their notebook (IP1, IP8). Data scientists would prefer that, "when the process is done, it automatically creates a notification" (IP15). At least this way, the data scientist knows that they can go back to the notebook (IP4, IP5, IP14). While this option is more convenient, data scientists reported reliability issues when scaling to big data.

Packaging as a standalone artifact. Because "notebooks encourage coding in cells, it's not software-engineering friendly" (IP15). During exploration and analysis, we found that data scientists often created quick and dirty code, but "code in production software is very different from code in the notebook" (IP15). To make matters worse, cells in the notebook don't have to execute linearly, and the desired execution order isn't stored explicitly with the notebook (IP6, IP15, FP2). For these reasons, "it's very difficult and painful to transition notebooks to production code without doing a full rewrite" (IP4, IP14, IP15).

SURVEY RESULTS

Our qualitative results from field observations and interviews show that data scientists' work is rife with pain points. But, which of the activities within these pain points are critical and obstruct notebook usage? Figure 1 and Figure 2 show a distribution of responses for importance and difficulty, respectively. The distribution of importance shows that all activities are at least moderately important (that is, median of responses were above 3) to data scientists.

Since all activities are challenging for some data scientists, we focus on *high-impact* activities when discussing design opportunities in the next section. These are activities that respondents marked (by median) as at least "difficult" and "important," across multiple types of notebooks. The four activities (and the associated pain points) under this criteria are: viewing and exploring history (archival), refactoring code in notebooks (manage code), scheduling long-running processes for automatic execution (notebooks as products), and easily deploying notebooks as products (notebooks as products).

Another lens through which we can prioritize activities is to look at *low-impact* activities. These are activities that are at least "important" and "easy" or "very easy." In other words, these activities are already well supported in computational notebooks, so notebook developers should avoid impacting these activities as they introduce new capabilities in their notebooks. These activities (and pain points) are: load (setup), explore (explore and analyze), and visualize data (explore and analyze), debug (manage). It's also notable that data scientists find both local and remote notebook environments important to their day-to-day activities.

DESIGN OPPORTUNITIES

The pain points in our taxonomy suggest both understudied research areas and design opportunities for improving computational notebooks for data scientists. We first discuss design opportunities for high-impact activities, followed by design opportunities in more niche data science scenarios. Encouragingly, notebook developers are already targeting some of these design opportunities—but support thus far is piecemeal, with data scientists having to shuffle across different computational notebook environments.

Refactoring

When compared with traditional integrated development environments, such as Visual Studio Code[7] or PyCharm[8], our data scientists report that proper coding assistance within notebooks is almost non-existent. Familiar features like autocompletion, refactoring tools, and live templates are often missing or do not function properly. Consequently, data scientists spend substantial time using online resources like Stack Overflow or documentation to help them code. Even when code snippets were available online, data scientists had to manipulate and wrangle these snippets to fit within their existing code, introducing unnecessary clerical errors in the process. A short-term pathway is for toolsmiths to incorporate the familiar coding assistance features of the integrated development environment and make them available within the notebook. A longer-term pathway is to leverage affordances unique to the computational notebook paradigm: the interplay between input code cells and graphical output cells enable new mixed-initiative experiences like programming-by-example [9, 25] and programming-by-demonstration [4, 26, 11, 16, 8] within the notebook environment. For example, consider a data scientist who writes some initial plotting code in an input cell that renders a plot in the output cell. With programming-by-demonstration, the data scientist would be able to directly manipulate the plot through graphical affordances, and these manipulations could automatically update the corresponding code.

In many ways, coding assistance is possibly even more important to data scientists than software engineers: for data scientists, coding is primarily a means to answer questions about their data, and not the core activity of interest. Languages such as R and Python are popular with data scientists because of their "swiss-army knife' capabilities, but the flexibility of these languages—for example, dynamic typing and reflection—make them difficult for static analysis tools to reason about [6, 7]. For researchers, it's important that we support the languages data scientists actually use, not the languages we *wish* they would use.

Deploying to Production

Deploying and adapting exploratory notebooks for production environments, such as customer-facing applications, requires data scientists to acquire expertise in a skill set that is well outside of their core, day-to-day responsibilities. While larger organizations have data engineering or software engineering teams to assist data scientists in deployment, in smaller organizations data scientists must take on these responsibilities themselves [1].

[7] https://code.visualstudio.com/
[8] https://www.jetbrains.com/pycharm/

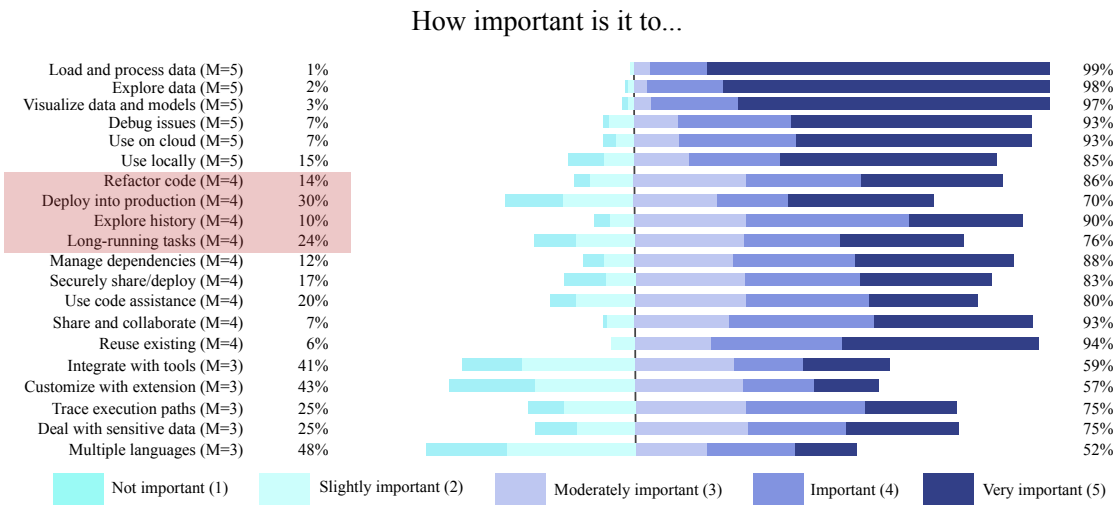

Figure 1: Distribution of importance for activities, measured from 1 (Not important) to 5 (Very important). The median importance for each activity is reported as M=median. Activities highlighted in red boxes are high-impact activities (median importance=4, median difficulty=4).

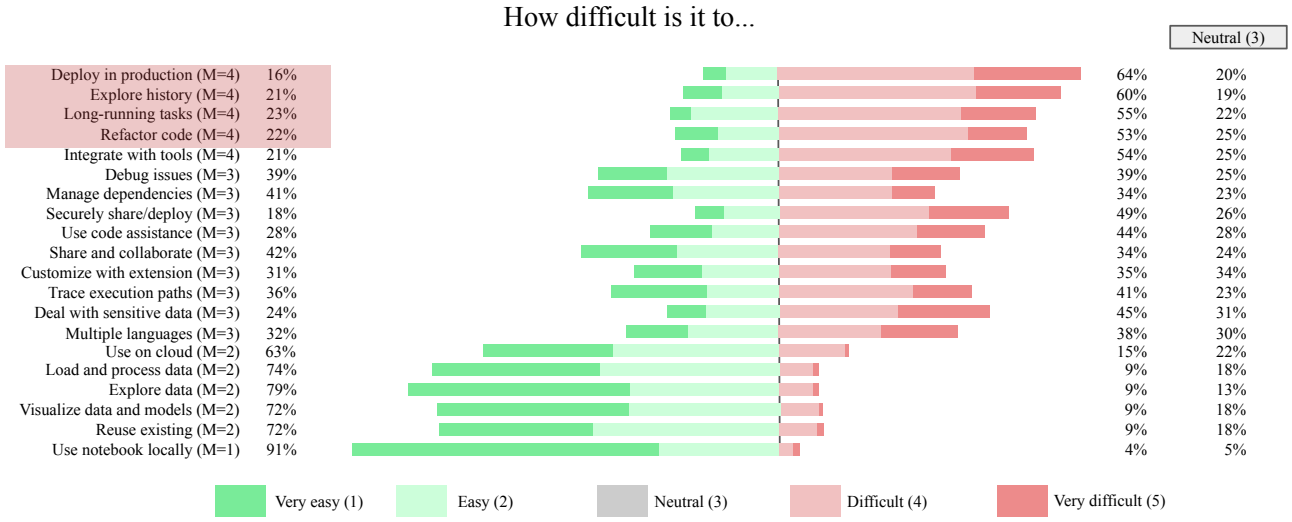

Figure 2: Net stacked distribution of difficulty for activities, measured from 1 (Very easy) to 5 (Very difficult). The net stacked distribution removes the neutral option, and shows the skew between Difficult (red bars) and Easy (green bars). The median difficulty for each activity is reported as M=median. Activities highlighted in red boxes are high-impact activities (median importance=4, median difficulty=4).

There are several opportunities to improve the data scientists' user experience and productivity. First, the programming languages for data science—such as Python or R—are not often the same programming languages that are used in production environments—such as C++ or C#. Automated translation tools or runtime environments are one opportunity to help mitigate these challenges. In circumstances when using the data science programming language is suitable, the data scientist must still clean up their notebook to remove unnecessary dependencies, dead-ends, and unused code in their exploration before exporting the notebook to a standalone script for execution in the production environments [14, 10]. Tools such as Papermill [27] are a step in the right direction: Papermill allows data scientists to directly deploy their notebook to a production environment. The tool enables "parameters" in the notebook; essentially, these parameters allow production systems to pass configuration options to the notebook, for example, the name of the data source to analyze. Still other promising directions can be found in computational notebook environments such as Databricks and Zeppelin. These environments provide higher-level abstractions and make default decisions about complex configuration options on behalf of

the data scientist, simplifying the user experience for them. In short, data scientists desire push-button approaches to productionize their notebooks.

History

Due to the affordance of out-of-order execution in computational notebooks, it can quickly—both unintentionally and *intentionally*—have an internal state that is different from the linear order of the notebook presented to the data scientist. This is one form of history-related pain point *within* a notebook. Without history support within the notebook, data scientists must painfully debug how their notebook entered the current state. As data scientists conduct exploration, build models, and evaluate multiple design options, they also introduce multiple notebooks into their environment, which they often manage in an ad-hoc way (for example, `experiment_current.ipynb` and `experiment_old.ipynb`). This is the other form of history-related pain point *between* notebooks. Without support for easily finding the right information across notebooks, data scientists end up reimplementing functionality or using stale data because they lose track of their notebooks.

In contrast to version control typically found in software engineering systems (`git`), preserving the history of artifacts is also important to data scientists. Novel innovations in the notebook space are required to support these unique requirements. For example, Head and colleagues [13] found that data scientists do not want to spend up-front effort in managing code versions or organizing their code, and that data scientists prefer automated approaches to version control management. Some notebook environments, such as Nextjournal[9], attempt to address this need by automatically and periodically versioning the notebook, data, and full runtime environment.

As computational notebooks use both text and visual medium, advanced tools and techniques are required that can easily differentiate changes (both text and visual) between different versions of notebooks. For example, Kery and colleagues [18] developed an experimental tool that complements existing version control systems and allows the data scientist to visualize the history of individual cells, code snippets, markdown, and outputs.

Finally, supporting annotations and comments in computational notebooks allows data scientists to catalog their thoughts and comments for decisions they've made in the notebook, essentially, a form of inline journaling.

Long-running Computations

The scale of data and computation increases as data scientists incorporate more demanding activities into their computational notebooks. Activities previously offloaded to standalone scripts or jobs, like working with large datasets, are now routinely conducted within the notebook itself. These activities are enabled by big data libraries that are directly accessible in Python, such as tensorflow and keras, as well as through data connectors that give data scientists access to large data repositories from within the notebook.

But large scale data means long-running computations, which often block operations in the underlying language kernel, creating tensions with the interactivity that data scientists expect from notebooks. For example, many data science libraries are synchronous, meaning that they block the data scientist from any other operations. And with streaming data—that is, data that is continuously generated by a data source—the program runs forever.

To support these activities, notebook developers must support scalable computations as a first-class design goal in notebooks. In other words, computational notebooks should maintain the benefits of exploration, interactivity, debugging, and visualization, irrespective of the size of the data.

Potential opportunities include *transaction* support, which allows the data scientist to abort long-running cells and revert to a safe state prior to executing the problematic cell. For streaming data, *reactive* notebooks such as BeakerX[10] and Tempe[11] automatically update computations as new data becomes available in the notebook. However, enabling interactivity and scaling requires careful design and introduces new challenges—in particular, these notebook models can exaggerate confusion data scientists already have with out-of-order execution.

RELATED WORK

How Data Scientists Use Notebooks

We found two studies categorizing how data scientists use notebooks. Kery and colleagues [20] studied coding behaviors in computational notebooks, focusing on how data scientists kept track of variations they explored. Rule and colleagues [34] looked at notebook structure, findings tensions between exploration and explanation in notebooks, and identifying reasons academic data scientists use notebooks—tracking provenance, reusing code, enabling replication, and presenting results. Our study differs in that we focus on *pain points* that professional data scientists have when working with computational notebooks.

The Jupyter organization conducted a survey to identify what respondents liked about Jupyter notebooks, and what needs remain unaddressed. Our study confirmed some of their findings [15] but disconfirmed others. For example, we also found that our respondents, for the most part, found it easy to explore and visualize in notebooks. However, our respondents did not find lack of debugging tools to be a substantial pain point. Our data scientists also reported that traditional version control solutions (like git) are *not* appropriate for computational notebooks, in contrast to the Jupyter survey. Moreover, our study systematically focuses on *why* certain activities are challenging across multiple computational notebook environments, using field observations and interviews.

Data Scientist Practices

Several studies investigated work practices of data scientists, although not specifically in computational notebooks. Kandel

[9] https://nextjournal.com/

[10] http://beakerx.com/

[11] https://www.microsoft.com/en-us/research/project/tempe/

and colleagues [17] described recurring pain points, outstanding challenges, and barriers to adoption for visual analytic tools. Muller and colleagues [24] described how scientists, scholars, engineers, and others work with their data. Wilson and colleagues [38] recommended several general practices for working in computational notebooks, such as modularizing code, but the exploratory nature of computational notebooks disincentivize several of these practices. Hannay and colleagues [12] found that most scientists learn to develop and use software informally from their peers. Kery and colleagues [2] described exploratory programming—such as in computational notebooks—as a key practice for data scientists; they explored further the behaviors and characteristics of this work. Wang and colleagues [37] proposed design implications to better support collaborative editing, as current synchronous editing features result in interference and imbalanced participation. And Kross and colleagues [22] studied how to teach data science in industry and academia.

Tools for Computational Notebooks

Researchers have developed some tools to improve the experience of computational notebooks, mostly around version control and history [18, 23, 19], cleaning and annotating code [34, 13], working with streaming data [5], and extending the notebook model to make visualizations more central [39]. Our study finds several additional opportunities for improving the user experience in notebooks, not yet addressed by existing research.

LIMITATIONS

Any empirical study has its strengths and weaknesses. Field studies excel in capturing how participants actually work, but this data has to be collected opportunistically. Researchers need to be present at the time a participant is engaging in the relevant activity, in our case, working on computational notebooks. Interviews, by contrast, can be scheduled a priori and are easier to perform. However, they depend on participants' memory and (sometimes) what participants *think* they should do and not necessarily what they *actually* do. Survey data is the easiest to collect, but it doesn't offer the richness of the other two empirical methods. A mixed-method study, like ours, helps balance the strengths and weaknesses of different methods.

Our study focused primarily on professional data scientists who use notebooks in their day-to-day work activities. Consequently, our findings may not generalize to other types of notebook users, such as machine learning enthusiasts, educators, end-user programmers, or infrequent notebook users. In addition, most of our participants primarily used Jupyter notebooks. Thus, some of the pain points may not apply to other notebooks, such as R Notebooks and Spyder, which have different architectures.

The importance and difficulty of our identified pain points are dependent on job roles, data scientist experience, and types of notebooks the data scientists primarily use. Our reported findings aggregated these factors, and any individual data scientist may have different priorities in how they perceive the importance and difficulty of these activities. In our survey, self-selection bias may have also influenced our results [3]: our respondents may feel more strongly about pain points in notebooks than the general population of data scientists.

Establishing validity in qualitative research is challenging due to several potential biases, including researcher bias, confirmation bias, and interpretive validity [28]. To reduce these issues, we employed three techniques. First, we calculated IRR for external raters who categorized participant statements to our pain points. Second, we conducted a survey to triangulate our interview and field study results. Within the survey, we asked participants in a short text response if there were any pain points we had missed: no new pain points were identified through these responses. Third, at the conclusion of our study, we conducted an informal focus group with both data scientists ($n = 3$) and computational notebook developers ($n = 2$, Jupyter and nteract) to identify any potential credibility issues.

The focus group identified some additional limitations in our study. Participants suggested that our pain points may be biased towards tabular ("rectangular") data, and that data scientists who work with images, audio, or natural language text may be underrepresented. For these domains, loading, processing, and exploring data may be far more difficult than our results suggest. Developers of notebooks identified that accessibility issues did not appear as a pain point in our study, although they routinely receive bug reports about these issues. They suggested that lack of keyboard shortcuts, text-to-speech support, and high-contrast themes, may impact productivity for some data scientists. Overall, however, the pain points identified in our study resonated with the focus group.

CONCLUSION

In this paper, we conducted a mixed-method study with data scientists using field observations, interviews, and a survey. In field studies and interviews, our data scientists reported a variety of difficulties when working with notebooks, and we synthesized these difficulties into a taxonomy of pain points. We validated the challenging activities that contribute to these pain points through a survey and found that supporting all of the activities were at least moderately important to data scientists, and that four activities—refactoring code, deploying to production, managing and working with history, and executing long-running tasks—were both broadly difficult as well as important to address, turning them into high-impact activities. Our findings suggest several design opportunities for researchers and computational notebook developers. Addressing these pain points can substantially improve the usefulness, productivity, and user experience for data scientists who work with computational notebooks.

Acknowledgements

We thank Audrey Au, Devesh Desai, Sumit Gulwani, John Lam, Natalia Morales, and Adriana Moscatelli for their help and feedback. This work is partially supported by the National Science Foundation under Grant Nos. 1560526 and 1815486.

REFERENCES

[1] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The bones of the system: A case study of logging and telemetry at Microsoft. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 92–101.

[2] M. Beth Kery and B. A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–29. DOI: http://dx.doi.org/10.1109/VLHCC.2017.8103446

[3] Jelke Bethlehem. 2010. Selection bias in web surveys. *International Statistical Review* 78, 2 (2010), 161–188.

[4] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.

[5] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James F Terwilliger, and John Wernsing. 2015. Tempe: Live scripting for live data. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 137–141.

[6] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: An IR for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. ACM, 55–66. DOI:http://dx.doi.org/10.1145/3359619.3359744

[7] Aviral Goel and Jan Vitek. 2019. On the design, implementation, and use of laziness in R. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[8] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

[9] Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering* 45, 137 (2016), 3–15.

[10] Philip J. Guo. 2012. BURRITO: Wrapping your lab Notebook in computational infrastructure. In *4th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*. USENIX.

[11] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 65–74. DOI: http://dx.doi.org/10.1145/2047196.2047205

[12] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software?. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE)*. IEEE, 1–8. DOI: http://dx.doi.org/10.1109/SECSE.2009.5069155

[13] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Article 270. DOI: http://dx.doi.org/10.1145/3290605.3300500

[14] Eric Horton and Chris Parnin. 2019. DockerizeMe: Automatic inference of environment dependencies for Python code snippets. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 328–338.

[15] Jupyter. 2015. Jupyter Notebook UX Survey. (2015). https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb

[16] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 3363–3372.

[17] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. 2012. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (Dec 2012), 2917–2926. DOI:http://dx.doi.org/10.1109/TVCG.2012.219

[18] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards effective foraging by data scientists to find past analysis choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 92. DOI:http://dx.doi.org/10.1145/3290605.3300322

[19] M. B. Kery and B. A. Myers. 2018. Interactions for untangling messy history in a computational notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 147–155. DOI: http://dx.doi.org/10.1109/VLHCC.2018.8506576

[20] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Article 174, 11 pages. DOI: http://dx.doi.org/10.1145/3173574.3173748

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

[22] Sean Kross and Philip J. Guo. 2019. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 1–14. DOI: http://dx.doi.org/10.1145/3290605.3300493

[23] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-versioning tool to support experimentation in exploratory programming. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 6208–6219. DOI: http://dx.doi.org/10.1145/3025453.3025597

[24] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How data science workers work with data: Discovery, capture, curation, design, creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Article 126, 15 pages. DOI: http://dx.doi.org/10.1145/3290605.3300356

[25] B. A. Myers. 1986. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 59–66. DOI:http://dx.doi.org/10.1145/22627.22349

[26] Brad A. Myers. 1998. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. ACM, 534–541. DOI: http://dx.doi.org/10.1145/274644.274716

[27] Netflix. 2018. Part 2: Scheduling Notebooks at Netflix. (2018). https://medium.com/netflix-techblog/scheduling-notebooks-348e6c14cfd6

[28] Anthony J. Onwuegbuzie and Nancy L. Leech. 2007. Validity and qualitative research: An oxymoron? *Quality & Quantity* 41, 2 (01 April 2007), 233–249. DOI:http://dx.doi.org/10.1007/s11135-006-9000-3

[29] F. Perez and B. E. Granger. 2007. IPython: A system for interactive scientific computing. *Computing in Science Engineering* 9, 3 (May 2007), 21–29. DOI: http://dx.doi.org/10.1109/MCSE.2007.53

[30] Fernando Perez and Brian E Granger. 2015. Project Jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September* 11, 207 (2015), 108.

[31] F. Perez, B. E. Granger, and J. D. Hunter. 2011. Python: An ecosystem for scientific Computing. *Computing in Science Engineering* 13, 2 (March 2011), 13–21. DOI: http://dx.doi.org/10.1109/MCSE.2010.119

[32] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier. 2014. The Jupyter/IPython architecture: A unified view of computational research, from interactive exploration to communication and publication. *AGU Fall Meeting Abstracts* (Dec. 2014), H44D–07.

[33] B. M. Randles, I. V. Pasquetto, M. S. Golshan, and C. L. Borgman. 2017. Using the Jupyter Notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. 1–2. DOI: http://dx.doi.org/10.1109/JCDL.2017.7991618

[34] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI)*. ACM, Article 32, 12 pages. DOI: http://dx.doi.org/10.1145/3173574.3173606

[35] Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers*. SAGE Publications.

[36] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature News* 515, 7525 (2014), 151.

[37] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction (CSCW)* 3 (2019), 1–30.

[38] Aruliah D.A. Brown C.T. Hong N.P.C. Davis M. Guy R.T. Haddock S.H. Huff K.D. Mitchell I.M. Plumbley M.D. Wilson, G. and B. Waugh. 2014. Best practices for scientific computing. *PLoS Biology* 12, 1 (2014), e1001745.

[39] Jo Wood, Alexander Kachkaev, and Jason Dykes. 2018. Design exposition with literate visualization. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2018), 759–768.

Restoring Execution Environments of Jupyter Notebooks

Jiawei Wang
Faculty of Information Technology
Monash University
Melbourne, Australia
jiawei.wang1@monash.edu

Li Li[α]
Faculty of Information Technology
Monash University
Melbourne, Australia
li.li@monash.edu

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

***Abstract*—More than ninety percent of published Jupyter notebooks do not state dependencies on external packages. This makes them non-executable and thus hinders reproducibility of scientific results. We present *SnifferDog*, an approach that 1) collects the APIs of Python packages and versions, creating a database of APIs; 2) analyzes notebooks to determine candidates for required packages and versions; and 3) checks which packages are required to make the notebook executable (and ideally, reproduce its stored results). In its evaluation, we show that *SnifferDog* precisely restores execution environments for the largest majority of notebooks, making them immediately executable for end users.**

***Index Terms*—Jupyter Notebook, Environment, Python, API**

I. Introduction

Jupyter notebooks—interactive documents that combine code, text, mathematics, plots, and rich media—have become a prime medium for scientists to document, replicate, and illustrate their findings. In contrast to a regular scientific paper, a notebook allows its writers to directly interact with data and code, updating tables and diagrams on the spot. This also extends to users, who can re-execute the notebook code, say with their own data or changes to the algorithms, and see how this affects the final results. This makes Jupyter notebooks one of the most promising tools to allow for widespread replication and reuse of research results.

What sounds good in theory need not be true in practice, though, and Jupyter notebooks are no exception. Recent studies [1] have shown that the vast majority of published Jupyter notebooks can only be read by users, but not re-executed. One reason is *incompleteness,* such as the raw data not being supplied; and there is not much users can do about this. However, there are also reasons for notebooks being non-executable that can be easily avoided. One reason is that notebook code cells can be executed interactively *in any order* (and data scientists happily do so); recent approaches [2] thus focus on restoring the actual order based on internal dependencies. Another important reason, however, is that Jupyter notebooks *depend on specific environments* in which they were created, such as specific libraries in specific versions.

In principle, Python code in notebooks provides *import* statements, which state the (external) modules are to be used. However, Python users install *packages,* not modules; and the names of imported modules may be different from the name of the package that provides them. Different versions of packages may provide different APIs; hence one has to determine compatible versions. Also, packages may depend on other tools or packages to be installed.

This is why Python (like other languages), in good Software Engineering tradition, has long introduced *explicit means to specify dependencies* between libraries and packages. Python package managers (e.g., pip and conda), for instance, expect Python packages to provide an explicit *list of dependencies,* stating which other packages need to be installed in which versions. Writers of Jupyter notebooks, however, are first and foremost data scientists and not software engineers [3]; hence, they neither know about principles of reusable software, nor would this be in their focus. Indeed, as we show in this paper, *around 94% of notebooks do not formally state or document dependencies;* among those who do, nearly 30% are not reliable. In consequence, users who want to execute published and complete Jupyter notebooks will very likely face errors of missing packages or incompatible versions.

In this paper, we introduce a novel approach to *automatically restore the experimental dependencies of Jupyter notebooks.* Our *SnifferDog* tool takes a Python Jupyter notebook and automatically detects which packages are required to reproduce notebook results. To this end, *SnifferDog* creates an *API bank,* a database which holds API information for each Python library (and each version). By analyzing the Python code embedded in the notebook, *SnifferDog* then determines library candidates that would be API compatible. *SnifferDog* then can automatically install the recommended dependencies and check if they allow the notebook to 1) be executed and 2) reproduce the original results stored in the notebook. When users thus apply *SnifferDog* on a notebook, they at least obtain a list of detected required libraries and their versions. If these are complete, the notebook can become executable; and in the ideal case, the notebook is shown to fully reproduce the original results. Striving for executability, reproduction, and considering library versions is also what sets *SnifferDog* apart from earlier, Python-specific approaches [4].

SnifferDog is efficient and effective: It finishes the analysis of 5,000 notebooks in 18,141.29 seconds (3.63 seconds per

[α]Corresponding author

1558-1225/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00144

notebook). In an experiment with 315 notebooks known to be executable, *SnifferDog* was able to automatically determine dependencies for over 90% of them.

The remainder of this paper is organized as follows. After providing background about Python packages and Jupyter notebooks (Section II), we make the following contributions:

- **A study on the prevalence of dependency issues in Jupyter notebooks** (Section III). In a preliminary study, we investigated causes that make Jupyter notebooks non-executable, with and without environmental dependencies.
- **A novel approach to restore dependencies of Jupyter notebooks** (Section IV). We present the design of our approach and its implementation in the *SnifferDog* prototype.
- **An evaluation of our approach** (Section V). We evaluate the effectiveness of *SnifferDog* on a variety of notebooks, showing that it precisely restores execution environments for the largest majority of notebooks.

After discussing related work (Section VI), we close with conclusion and future work (Section VII).

II. Background

We start with discussing background knowledge, including Python libraries and Jupyter notebooks.

A. Python Libraries

Python is well-known for its immense ecosystem, providing more than 200,000 third-party packages (also known as libraries) to developers. Such Python libraries need to be locally installed onto developers' implementation environment before being accessed. The Python Packaging Authority team officially maintains a standard package management tool called *pip,* which allows users to install these libraries from different sources (PyPI) [5]. In addition to package management systems, Python developers can also install a library from its source code project.

Figure 1 illustrates a typical code structure example of a Python library. A Python file named *setup.py* installs the library locally. However, this file will not install the library's environmental dependencies and, hence, requires its users to fulfill them beforehand. A *top-level package* giving the library its name (i.e., pandas) is stored in the same level of *setup.py*. Such a *package* in Python is a directory that contains a specific file named *__init__.py* responsible for initializing the package. Python packages provide a way to structure Python's module namespace, offering an easy means for library users to access its APIs.A *module* in Python is a specific term used in Python to specify Python's source code (i.e., files containing Python definitions and statements). Each Python file represents a Python module; for instance, *setup.py* defines a Python module named setup. Directories under the top-level package are the library's *sub-packages.* Similarly, the Python files under sub-packages are deemed as sub-modules. For example, as shown in Figure 1, the Python file *base.py* is defined as a sub-module named *base* in sub-package *pandas.io.excel*.

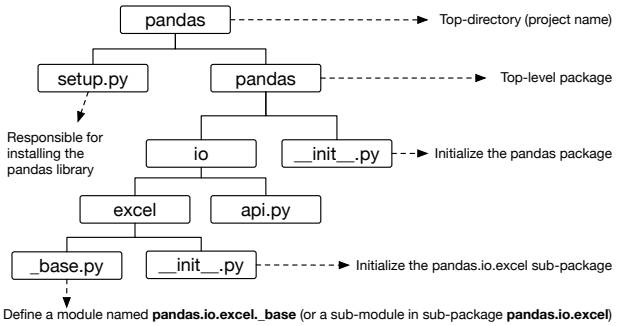

Fig. 1: A typical code structure example of Python libraries. This partial code structure is extracted from a popular Python library called Pandas.

In each Python module (or Python file), a set of methods can be declared and implemented. These methods can be accessed by other modules (or module users) and hence are referred to as *APIs*. For example, the *pandas.io.excel.base* module contains an API called *read_excel()* with the *fully qualified name* being *pandas.io.excel._base.read_excel()*). When Python libraries evolve, their declared API sets will likely be updated. In this work, we will leverage this information to implement *SnifferDog* so as to infer environmental dependencies for Python Jupyter notebooks.

B. Jupyter notebooks

Jupyter notebooks are sequences of *cells*, which either contain *text* (in Markdown format) or *executable code* (and its results). In *text cells,* authors describe (using Markdown and HTML for rich formatting) the objective of the notebook and the rationale behind the code presented in the following cells. In *code cells,* authors write actual programming code, most frequently Python code. Figure 2 presents a typical example of a Jupyter notebook, containing three text cells and six Python code cells.

Each code cell can be directly executed by the underline Jupyter engine, which provides the necessary computational environment such as library dependencies. The code cells in Jupyter notebooks can be executed in any order (producing errors if its prerequisites are not satisfied). After one cell is executed, Jupyter will assign an execution order aligning with its execution order. For example, the first executed cell will be marked as "In [1]", while the fourth executed cell will be marked as "In [4]". Cells can be repeatedly executed. In such a case, the latest execution counter will overwrite the previous one..

Looking closely at Figure 2, we see that the last code cell is executed before the fourth and fifth code cells. Astute readers may have also observed that there is no information (i.e., "In [5]") indicating the fifth executed code cell. This is because when code cells are repeatedly executed, the original execution counter will be overwritten by the last execution counter—called a *skip* of execution counter. Skips make it

hard to reproduce the original outputs of a notebook because the skipped execution counters are not recorded at all [2].

If the execution of a code cell generates an output (text or pictures such as diagrams), the output will also be recorded and displayed in the notebook. In Figure 2, a histogram at the end is the output of the last code cell.

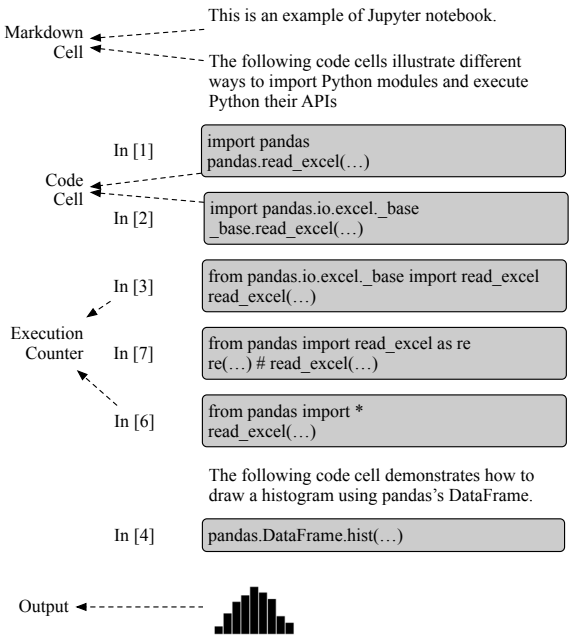

Fig. 2: An example of Jupyter notebook.

III. Preliminary Study and Motivation

In 2019, Pimentel et al. presented a large-scale empirical study [1] on the quality and reproducibility of Jupyter notebooks. In this study, the authors looked into 1,159,166 notebooks collected from GitHub, among which only 149,259 (roughly **12.9%**) of them were provided with module dependency information describing how the notebooks' environmental dependencies should be set up. In other words, the vast majority of existing notebooks in the community do not provide sufficient information such that notebook users could execute and replicate them. Since easy replication is one of the promises of Jupyter Notebooks, there is a need for dependable automated approaches to infer environmental dependencies for Jupyter notebooks.

How serious is this problem? We have conducted a lightweight replication study of Pimentel et al.'s work on recent Jupyter notebooks. We limit our replication study to replicating the executibility of notebooks when supplying dependencies provided by the notebook authors to identify the main causes for non-reproducibility and thus specifically address execution environments. To fulfill this purpose, we propose to answer the following research questions

- **RQ1:** To what extent do (public) Jupyter notebooks provide environment setup information?
- **RQ2:** How useful is dependency information in helping notebook users configure the execution environment?
- **RQ3:** Does the provided environment information help notebook users to execute and reproduce the notebooks? If not, what are the root causes making them non-executable?

To answer these research questions, we have collected a dataset consisting of notebooks with and without experimental setup information. Our source for notebooks is *GitHub,* one of the world's leading software repository hosting platforms. We randomly downloaded 100,000 notebooks from GitHub.

Table I summarizes our study results. Among 100,000 notebooks, less than 6% of them (or 4.74%, 1.15%, and 0.12% respectively for the three selected criteria) have been provided with environmental dependency information[1]. This rate is even lower than the rate (calculated similarly) reported by Pimentel et al. two years ago.

Note that prior work by Pimentel et al. has investigated three sources for notebook environment setup information: (1) requirements.txt, (2) Pipfile, and (3) setup.py. As discussed previously, *setup.py* is usually used to locally install a Python library from the source. It is not responsible for installing library dependencies. After manually investigating notebooks that use *setup.py,* we confirm that *setup.py* is indeed not relevant to the environment setup of Jupyter notebooks. Therefore, we exclude the third criteria *setup.py* for this study. Furthermore, when conducting the previous manual analysis, we additionally find that notebook contributors may provide environmental setup information through Anaconda (e.g., via *environments.yml*). As a result, in our study, we replace the third criteria *setup.py* with Anaconda environments.

TABLE I: Distribution of notebooks being provided with environmental dependency information w.r.t. the selected three criteria.

	requirements.txt	environments.yml (Anaconda)	Pipfile	Total
Notebooks	4741	1146	117	5826
Notebooks (≥3.5)	2923	868	77	3740
Installable	2064	563	77	2646
Executable	518	207	14	725

RQ1: To what extent do (public) Jupyter notebooks provide environment setup information?

Among 100,000 notebooks, only less than 6% provide environmental dependency information for helping users execute their notebooks.

For the 5,826 notebooks that *have* been provided with environmental dependencies, we further check how *reliable* these are. To this end, we implemented scripts to automatically install such dependencies, using Anaconda [6] to create individual environments for each of the aforementioned Jupyter

[1]Some notebooks may provide two types of information for helping users setup the execution environments. For example, there are 101 notebooks contain both *requirements.txt* and Anaconda information.

```
## requirements.txt ##
pandas  # Without specifying versions
scipy   == 1.17.5  # Must be version 1.17.5
sklearn >=0.23.0.  # Minimum version 0.23.0

## Conda environment.yml ##
name: env_name
dependencies:
- numpy=1.11.1=py35_0
- openssl=1.0.2h=vc14_0
- pandas=0.18.1=np111py35_0

## Pipfile ##
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
pandas = "*"
sqlalchemy = ">= 1.3.0"
```

Fig. 3: Example of the requirement files, Anaconda environment.yml and Pipfiles.

notebooks. After that, we leveraged a tool named *nbconvert*[2] to evaluate the execution of notebooks. Since *nbconvert* in Python 3.5 or its lower versions are no longer supported by the official Jupyter team, we had to exclude 2,086 notebooks that cannot be analyzed by our scripts.

After automatically installing the remaining 3,740 notebooks, we resort to their *logs* to check whether the installations are successful or not. Based on our observation, when an installation fails, it will contain one of the following three messages: (a) "InstallationError" occur as the runtime exception, (b) "ERROR:", and (c) "cannot find a version for". Of the 3,740 notebooks, 1094 notebooks failed to have their dependency information installed, giving a failure rate of 29.25%.

RQ2: How useful is dependency information in helping notebook users configure the execution environment?

In our evaluation, 29.25% of the environmental dependency information provided by notebook contributors was found to be unreliable and/or insufficient.

For the notebooks that we can successfully install their environmental dependencies, we go one step deeper to check if the installed dependencies are *adequate* to support the execution of the notebooks. Recall that notebook code cells can be executed in any order and can be repeatedly executed (resulting in skipped execution counters). Hence, it is practically impossible to infer the actual execution order initially conducted by the notebook contributor. In this preliminary study, we simply execute the code cells top-down. We believe this order reflects the natural flow indicating how its contributor has attempted to implement the notebook. The execution time is set 10 minutes for every notebook.

[2]https://github.com/jupyter/nbconvert

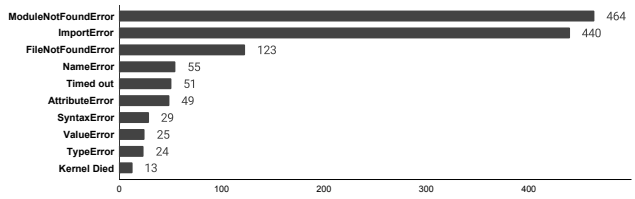

Fig. 4: Top 10 runtime exceptions from executions of notebooks whose dependency files can be successfully installed.

The last row of Table I presents the number of notebooks (i.e., Installable ones) that can be executed without errors. In other words, among 2,646 notebooks that have been provided with installable dependency information, 72.6% of them cannot be successfully executed following the straightforward top-down execution strategy.

There are various reasons causing execution of these notebooks to fail. Indeed, on the one hand, the provided dependency information may not be perfectly reliable, resulting in dependency-related errors. On the other hand, even if the environmental dependencies are correctly set up, notebooks themselves may contain implementation errors that can also lead to runtime exceptions.

Figure 4 enumerates the top-10 errors summarized from the unsuccessful notebooks. The fact that the top-ranked errors are related to environmental dependencies shows that environmental dependency seems to be the primary reason causing execution errors of the aforementioned notebooks. The top-2 ranked errors (i.e., module not found and import error) are indeed caused by the inadequate runtime environment, where the imported modules cannot be located [7]. In fact, notebooks linked to these two errors have accounted nearly half (24.15% and 22.90%, respectively) of the aforementioned unsuccessful notebooks.

```
# Example (1): ModuleNotFoundError
# from GitHub project  BenjaminBossan@mink
# requirements.txt
scikit-learn
...

# Module 'sklearn.grid_search' was removed since
    scikit-learn version XXX
Error: No module named 'sklearn.grid_search'


# Example (2): ImportError
# From GitHub project stargaser@astrodata2016
# requirements.txt
astroquery
...

# scale_image API was removed from module
    'astropy.visualization' since astroquery
    version XXX
Error: cannot import name 'scale_image' from
    'astropy.visualization
```

Listing 1: Two real-world examples suffering from runtime errors due to unmatched library versions.

Among various reasons causing notebooks failing to be successfully executed, we further look into some of the failures related to the top-2 types of errors and find that a significant amount of failures are due to different versions of libraries are installed. Indeed, when providing environmental dependencies (cf.Figure 3), notebook contributors are not required to specify the *exact versions* of the dependent libraries. As a result, the unmatched library versions might be installed and hence lead to runtime errors. Listing 1 demonstrates two of such examples (respectively for *ModuleNotFoundError* and *ImportError*) obtained from real-world notebooks. Due to the fast-evolving nature of software systems such as Python libraries, certain APIs might be deprecated and subsequently removed. If the wrong library versions are used, client applications, if not changed, will likely be subject to compatibility issues and hence result in runtime errors. Therefore, we argue that when specifying the environmental dependencies for Jupyter notebooks, it is essential to also *clearly specify the required versions of required libraries.*

RQ3: Does the provided environment information help notebook users to execute and reproduce the notebooks?

For 72.6% of notebooks, the provided dependencies are not sufficient for re-executing them without errors.

IV. *SnifferDog*

In this section, we present our approach to automatically inferring environmental dependencies for Python Jupyter notebooks, implemented in our *SnifferDog* prototype. Figure 5 summarizes the working process of *SnifferDog*, including mainly three modules: (1) Library API Mapping, (2) Library Identification and API Standardization, and (3) API Usage Analysis. We now detail these three modules, respectively.

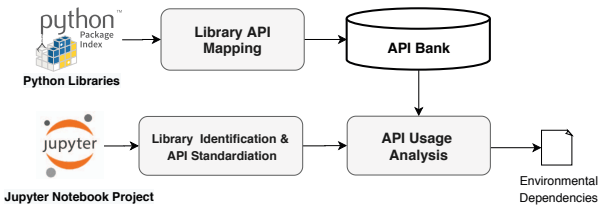

Fig. 5: The working process of *SnifferDog*.

A. Problem Statement

Before providing the details of our approach, we formally define the problem that we plan to address in this work. At first, we need to pre-build an API bank $\mathcal{L} = \{L_1^v, L_2^v, L_3^v, ...\}$ that records a large number of popular Python libraries' API sets, in which L_j^v stands for the set of APIs defined in library L_j at version v. Then, given a Python Jupyter notebook N as an input, we need to precisely parse its accessed API set $P = \{A_1, A_2, A_3, ...\}$, where A_i is an API used in P. After that, based on the pre-built API bank $\mathcal{L}$, the goal of this approach is hence to identify a set of libraries L that fulfill the following constraints: (1) $L \subset \mathcal{L}$, and (2) $\forall A_i \in P$, $\exists L_j^v \in L$ that $A_i \in L_j^v$.

B. Library API Mapping

The first module, *library API mapping*, is not directly related to the working process of analyzing concrete Jupyter notebooks but plays an independent step in building the core infrastructure of our approach. The output of this module will be an API bank that provides an extensible (and on-growing) database recording mappings from popular Python libraries to their APIs.

Given a Python library planned to be included in the API bank, this module first builds a directory tree following the file and directory composition of the library, aiming at providing a clear way to referring library APIs (e.g. from top-level package to the leaf module). In this directory tree, Python packages (or sub-packages) are represented by non-leaf nodes and Python source code files are represented by leaf nodes. Figure 1 presents such an example, representing a partial code structure tree of the popular Python library *pandas*. This module then builds Abstract Syntax Tree (AST) trees for each leaf node (or Python file) and traverses the trees to locate public functions, including their positional and keyword parameters. The output of this step can already build a mapping from the library (in a certain version) to its defined APIs.

Unfortunately, this approach may overlook certain API usages. Applying it to *pandas* (Figure 1), the API *read_excel()* (defined in the *base.py* module) can be referenced via its full qualified name *pandas.io.excel._base.read_excel()*, or *_base.read_excel()* (or *read_excel(()*) if module *pandas.io.excel._base* (or the API itself *pandas.io.excel._base.read_excel*) is imported, as respectively shown in the second and third code cells in Figure 2. However, as demonstrated in the first, fourth, and fifth code cells Figure 2, API *read_excel()* could be invoked via another forms such as *pandas.read_excel()*, i.e., it can be directly imported from the *pandas* module despite it being defined in the *pandas.io.excel._base* module.

This ambiguity is part of the complicated Python import mechanism, which has been implemented in a transitive manner. Let $X \xrightarrow{f} Y$ be importing API f from module X to module Y via statement *"from X import f"* in the source of Y. Transitivity enables $X \xrightarrow{f} Y$, if $Y \xrightarrow{f} Z$ and $X \xrightarrow{f} Z$. This feature, offered by Python runtime, has been frequently leveraged by many Python libraries to provide simplified means for users to access their APIs since it can transparently shorten the full qualified API names.

To resolve this feature, while parsing Python source code, we further conduct an *import-flow* analysis to find all the possible alternatives (or aliases) of directly defined APIs. Take the API *read_excel* again as an example, regarding the simplified source code shown in Figure 6(a), the *import-flow* analysis would lead to the following two flows.

pandas.io.excel._base $\xrightarrow{read\_excel}$ *pandas.io.api*

pandas.io.api $\xrightarrow{read\_excel}$ *pandas*

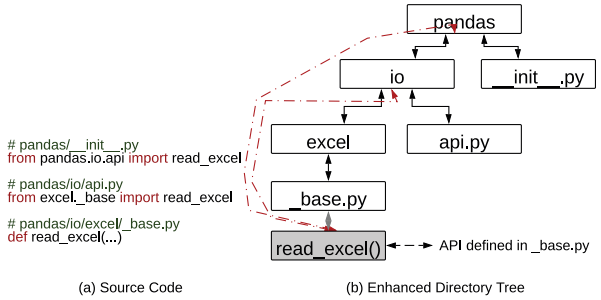

Fig. 6: An example of enhanced directory tree for Python library pandas.

By taking transitivity into consideration, we could further deduce the following flow.

$$pandas.io.excel.\_base \xrightarrow{read\_excel} pandas$$

Subsequently, at the end of this module, we further leverage these inferred and deduced import flows to enhance the directory tree initially constructed for the library (cf. Figure 6(b), before recording them into the API bank. The enhanced directory tree allows us to generate a complete list of APIs for each library integrated into the API bank.

C. API Identification and Standardization

As shown in Figure 5, the second module *API identifiation* focuses on analyzing *Jupyter notebook* projects (rather than Python libraries as that being targeted by the first module) to identify and standardize library API usage. Python code in notebooks can access (1) methods available in *local modules* that are often developed by the notebooks' contributors, (2) *Python standard methods* (i.e., often known as system APIs) that are provided by the core Python modules, and (3) *library methods* (i.e., often known as library APIs) that are developed by third parties and should be imported from external resources. In this module, we are only interested in the third type of methods, namely library APIs. To distinguish library methods from local modules and system APIs, we consider all the methods that are not defined locally and are not from Pythons' system APIs as library APIs [8]–[13], [13]–[16].

Following the same approach implemented in the *library API mapping* module, we first build AST trees for the notebook's Python code and then traverse the trees to identify library APIs. After that, this step goes one step deeper to expand the identified APIs to their fully-qualified names, based on the information extracted from the *import* and *from-import* statements. For example, for the API call statement *_base.read_excel()* in the second code cell in Figure 2, the standardized API will be *pandas.io.excel._base.read_excel()*. If the identified API is an alias, i.e., *re()* in the fifth code cell in Figure 2 defined by statement *from pandas import read_excel as re*), we will further replace it with its actual name while conducting the API standardization step. The standardized version will hence be *pandas.read_excel()*. Observant readers may have observed that our approach will lead to two full-qualified names for the same API *read_excel()*. Indeed, at this stage, it is non-trivial for our approach to be aware of that by simply analyzing the notebook code. We hence consider them as two independent APIs. Nevertheless, as discussed in the previous subsection, both of these two full-qualified API names will be recorded in the API bank thanks to the *import-flow* analysis. Hence, any imprecision in the analysis will not impact the overall precision of our approach.

Moreover, Python code may involve instances of library classes that are initialized by calling constructor methods. The APIs invoked by those instances should also be appropriately identified and expanded. However, in Python, there is generally no syntax level difference between initializing constructor methods and accessing standard methods. Therefore, additional efforts are needed to distinguish them and thereby to allow the identification of classes' instances and their accessed APIs. As an example, consider the code snippet in Listing 2:

```
1 from x import y
2 m = y()
3 m.fun()
```

Listing 2: An example of qualifying object member function calls.

In Listing 2, our approach will first identify that *m.fun* is a member function call and then trace back to its construction call *m = y()*; hence method *fun()* is an API in module *x*. Subsequently, the fully qualified name of this API will be *x.y.fun*.

D. Library Usage Analysis

The last module of *SnifferDog*, *library usage analysis* is straightforward. Based on the second module's outputs (i.e., a set of APIs), this module queries these APIs against the API bank to find possible library candidates who have provided these APIs. Normally, because each API has been provided with a full-qualified name, the API bank can often precisely locate its belonging library. The query output will hence be multiple releases (or versions) of the same library. By integrating the query results of all the identified APIs, the objective of this module is hence to find a (minimal) list of libraries and their (maximum) version ranges that cover all the identified APIs leveraged by the input notebook. *SnifferDog* will then produce its output in common formats that describe Python environmental dependencies, such *Pipfile* or *requirements.txt*.

V. Evaluation

To evaluate the effectiveness of *SnifferDog*, we address the following research questions:

- **RQ4:** Is *SnifferDog* effective in mapping Python libraries to their APIs?
- **RQ5:** How accurate is *SnifferDog* in inferring environment dependencies for Python Jupyter notebooks?
- **RQ6:** To what extent can *SnifferDog* assist users in reproducing Jupyter notebooks?

A. Experimental Setting

Dataset of Jupyter notebooks. Recall that the goal of this work is to automatically infer environmental dependencies for Jupyter notebooks so as to help users execute and reproduce notebook outputs. To evaluate if our approach can achieve this objective, we resort to the approach introduced by Pimentel et al. [1] to collect 100,000 Jupyter notebooks from GitHub to fulfill our experiments. GitHub is the world's leading software development platform hosting millions of software repositories. The 100,000 notebooks are retrieved from GitHub projects containing files with Jupyter notebook *.ipynb* formats and declaring Python as their programming language.

Dataset of Selected Python Libraries. Recall that the API bank of our approach is built based on existing libraries, and it can be easily extended to include more libraries. Generally, the more libraries considered, the more comprehensive the API bank will be, and subsequently, the more precise and sound results *SnifferDog* can achieve. Since we aim at generating dependencies for as many notebooks as possible, we start by selecting the most popular 1,000 modules imported by the aforementioned 100,000 Jupyter notebooks. We then leverage PyPI, the official Python package index, to query the installation wheel files which contain the source code of library implementation these selected modules. Because several modules may belong to the same library, or some modules have not yet been indexed by PyPI, we can only locate 488 Python libraries (with 17,947 different releases) for the selected top-1000 modules. Therefore, in this work, we leverage 488 distinct Python libraries with 17,947 releases to construct the API bank.

B. RQ4: Effectiveness of API Bank

In this research question, we are interested in evaluating the usefulness of the API bank. From the selected 488 Python libraries, the library API mapping module extracts 1,013,718 APIs to fill the API bank. Figure 7 illustrates the distribution of the number of APIs in each selected library, giving median and mean values at 321 and 2,281, respectively, after excluding outliers.

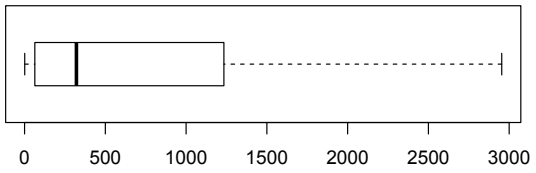

Fig. 7: The distribution of the number of APIs per library across all its version after removing outliers.

Towards evaluating the correctness of the constructed API bank, we resort to a manual process to check if these APIs are correctly recorded in the API bank. To this end, we randomly selected 166 APIs from the API bank to be manually validated. The number of selected APIs is decided by an online Sample Size Calculator [17] with a confidence level at 99% and a confidence interval at 10. For each of the selected APIs, we manually check it against its source code and find that 164 of them are correct results, giving a precision of 98.8% for our API bank construction approach.

In addition to the aforementioned manual investigation, we further resort to a *dynamic testing approach* to evaluate the correctness of the constructed API bank. Giving a mapping from a library version to its APIs, when the library (with the given version) is installed, all its APIs should be able to be imported. To this end, we implement a prototype tool to fulfill this process automatically. Specifically, we first randomly select 20 libraries, accounting for in total 3,982 APIs, from the API bank, and install them, respectively. For each of the installed libraries, we then extract all of its recorded APIs from the API bank and conduct runtime import testings to check whether these APIs can be imported at runtime. Among the 3,982 considered APIs, only 252 of them fail to be imported in our experiment, leading to a success rate of 93.6%. After analyzing the traceback information of import errors, we find that most of such failures are related to missing dependencies that are further required by the libraries under evaluation.

RQ4: (Effectiveness of library mapping) Is *SnifferDog* effective in mapping Python libraries to their APIs?

The API bank constructed by the Library API mapping module is precise: 98.8% of APIs are correctly extracted; 93.6% can be successfully imported.

Among the 1,013,718 APIs inferred from the 488 libraries, 686,915 of them could further introduce compatibility issues to their client applications (if incorrect library versions are installed), resulting in, for example, module not found errors and import errors. The incompatible APIs include 543,387 (53.60%) newly added APIs after the first libraries' releases, 345,234 (34.06%) removed APIs compared to the libraries' latest versions, and 58,594 (5.78%) APIs have their parameters changed over the libraries' evolution. Figure 8 further presents the distribution of newly added, removed, and updated APIs in each of the considered libraries, respectively.

Given the fact that 67.76% of the APIs (including added, removed, and updated ones without duplication) may introduce compatibility issues, there is a strong need to also infer the correct versions of the dependent libraries when inferring the environmental dependencies for Jupyter notebooks, Our API bank records the detailed evolution changes of considered libraries and is designed to infer not only the dependent libraries, but also their correct versions.

RQ4 (Usefulness of API bank) Is *SnifferDog* effective in mapping Python libraries to their APIs?

In our evaluation, more than half of the library APIs were added, removed, or updated at some point in the libraries' life cycles. This underlines the need to check for compatible library versions, as *SnifferDog* does.

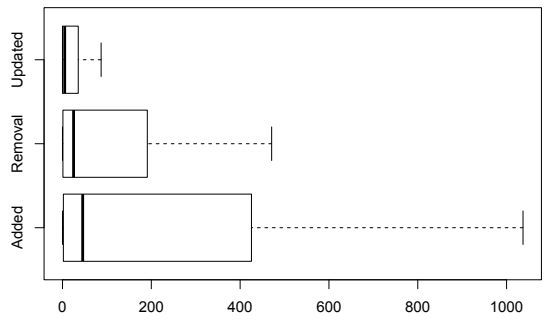

Fig. 8: The median values for the number of added and removed and updated APIs are 45, 25 and 5 respectively after excluding outliers.

C. RQ5: Effectiveness of SnifferDog

Let us now evaluate the effectiveness of *SnifferDog* in inferring environmental dependencies for Jupyter notebooks. We evaluate the effectiveness through one *in-the-lab* and one *in-the-field* experiment.

1) In-the-lab experiment: Recall that our preliminary study has identified 725 notebooks that are (1) provided with installable required dependencies, and (2) demonstrated to be executable after the provided dependencies are installed. We hence take these 725 notebooks as the ground truth to fulfill our in-the-lab experiment (because these notebooks are known executable). Unfortunately, 385 (out of the 725) notebooks have accessed libraries that are not yet considered by the current API bank (constructed based on around 488 libraries). Therefore, we have to exclude them from the ground truth. Our final ground truth is hence made up of 340 Jupyter notebooks and their required libraries.

For the 340 notebooks, we then apply *SnifferDog* to automatically generate experimental dependencies for them. After that, we follow the same approach (as discussed in Section III) to automatically install the generated libraries and execute the corresponding notebooks. Experimental results show that *SnifferDog* can successfully generate installation requirements for 315 (92.65%) notebooks, among which 284 are successfuly executed, giving a recall rate at 83.52%.

The installation failures are mainly related to library compatibility issues brought by the selected Python version (which is usually not provided by notebook contributors) and the underline Python setuptools [18]. For the 31 non-executable cases, our manual investigation reveals that the failures (8 ImportError, 7 ModuleNotFoundError and 16 other type of rumtime errors) are caused by inaccurate version constraints yielded by *SnifferDog*.

RQ5: (in-the-lab) How accurate is *SnifferDog* in inferring environment dependencies for Python Jupyter notebooks?

In a lab setting, *SnifferDog* is effective in automatically inferring execution environments for Jupyter notebooks, successfully generating installation requirements for 315/340 (92.6%) of notebooks. 284/315 (90.2%) of notebooks could be executed automatically.

2) In-the-field experiment: In this setting, we randomly select 5,000 notebooks and launch *SnifferDog* to generate execution environments for them. *SnifferDog* completes its analysis in 18,141.29 seconds, or 3.63 seconds per notebook on average.

We now check to which extent the generated environments support the execution of notebooks. To reduce human influence to a minimum, we restrict ourselves to a subset of notebooks to fulfill this purpose as it is time-consuming to evaluate a notebook, which involves installing all the dependencies and executing all of its code cells. To this end, we apply the following inclusion criteria to retain notebooks that (1) have been provided with pre-defined dependencies, which, however, cannot support their executions, and (2) are within the capacity of our API bank. This gives us 722 notebooks for the in-the-wide experiment.

Among the 722 notebooks, *SnifferDog* can successfully generate installable dependencies for 667 of them, among which 223 of them can further lead to successful executions of the corresponding notebooks.

Note that over half of the notebooks remain non-executable. Why is that so? Our manual analysis reveals the following two main reasons (apart from issues raised by notebooks' code qualities).

- **Reason 1:** The majority of notebooks fail to be executed because of the existence of so-called *optional dependencies,* which are not directly accessed by the notebooks (hence overlooked by *SnifferDog*) but are required by the notebooks' directly dependent libraries.
- **Reason 2:** A number of failed notebooks are due to the usage of *magic functions*, a special Jupyter notebook feature allowing the access of Python modules without following Python's syntax [19]. At the moment, magic functions are simply ignored by *SnifferDog*.

Moreover, for the failed notebooks, we further look into their error messages and compare them against that outputed from executions with their own dependencies (considered as the Baseline). In this experiment, only such notebooks that fail on both sides are considered. Figure 9 presents the comparison results. Clearly, the number of errors related to the execution environment for *SnifferDog* is significantly smaller than that of the Baseline. Oppositely, *SnifferDog* leads to more errors related to the code quality of the notebooks (e.g., FileNotFound, NameError, or HTTPError) compared to that of the Baseline. This experimental result shows that, while the notebooks fail to be executed in both environmental settings, the settings resulted from *SnifferDog* are more likely to be

correct than its counterpart (new runtime exceptions can be further triggered when dependencies are supplied, e.g., for FileNotFoundError, the file is not provided). Hence, even if the notebook is not executable yet, the dependencies produced by *SnifferDog* can assist users in getting closer to their goal.

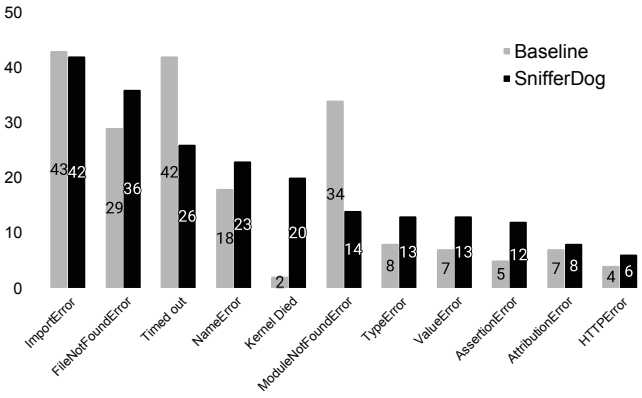

Fig. 9: Top-10 runtime error types resulted from 255 notebooks that can be not executed by environments recommended by *SnifferDog* and their original ones.

RQ5: (in-the-field) How accurate is *SnifferDog* in inferring environment dependencies for Python Jupyter notebooks?

Applied on random notebooks, *SnifferDog* is effective in automatically inferring execution environments. When notebooks remain non-executable, dependencies inferred by *SnifferDog* help users to get them closer to the goal.

D. RQ6: Reproducing Jupyter notebooks

In the last research question, we evaluate if *SnifferDog* helps users not only execute, but *reproduce* Jupyter notebooks (i.e., achieving the same outputs as that recorded originally in the notebooks). To resolve the problem of cell ordering, we make use of a prototype tool called Osiris that restores the order in which cells are to be executed [2].

For our evaluation, we resort to the 507 notebooks that have been demonstrated to be executable previously. We attempt to reproduce these 507 notebooks through the following two experiments:

- **Experiment 1: Osiris (baseline).** We create execution environments for different Python versions and install all default packages (including more than 200 library packages). Then We apply Osiris in this environment to analyze the 507 notebooks.
- **Experiment 2: Osiris (*SnifferDog*).** In this experiment, we use the dependencies recommended by *SnifferDog* to set up the environment, and we launch Osiris in this environment on the same 507 notebooks.

Figure 10 presents the experimental results. Interestingly, although with over 200 popular Python libraries installed in the execution environment, around one-fourth of notebooks cannot be fully executed. Compared to the default environment, the failure rate decreases to less than 1% when the execution environment is set up based on the outputs of *SnifferDog*[3]. Regarding reproducibility, Osiris reports that, with the help of *SnifferDog*, 42 additional notebooks could be reproduced compared to the 323 notebooks reproduced with the default Osiris configuration. These experimental results show that our approach is indeed effective in helping users execute and reproduce Jupyter notebooks.

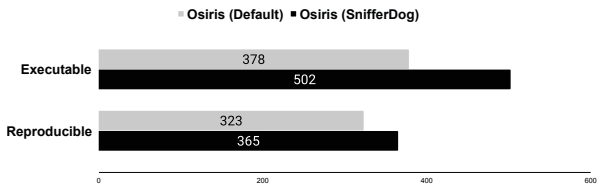

Fig. 10: Running Osiris in a default environmental setting "Osiris (Default)" vs. running Osiris in a *SnifferDog*-generated environmental setting "Osiris (*SnifferDog*)".

RQ6: To what extent can *SnifferDog* assist users in reproducing Jupyter notebooks?

Compared to a default environment setting, environments restored by *SnifferDog* makes significantly more notebooks executable and reproducible.

E. Threats to Validity

The experimental findings may suffer from various threats to validity. The notebooks selected in this work may not be representative. We attempt to mitigate this threat by starting from 100,000 real-world Jupyter notebooks. We also resort to manual analysis to summarize the execution errors as well as confirm some of our experimental results. Such manual processes are known to be error-prone. We have cross-validated the results to mitigate potential errors.

VI. Related Work

We now discuss the closely related works from three aspects: Restoring Execution Environments, Studies on Jupyter Notebooks, and Python dependency analysis.

A. Restoring Execution Environments

The most related work to ours is *DockerizeMe* [4], inferring environment dependency configurations for Python code snippets collected from GitHub Gist. *DockerizeMe* is implemented based on a pre-acquired knowledge base of known Python packages using static analysis, which is similar to the approach of ours. However, in addition to static analysis, it further leverages dynamic analysis to achieve its purpose, being able to achieve 30% improvement in reducing *ImportError*

[3]Ideally, there should be no failed cases because those notebooks have been demonstrated to be executable with the same environment. However, in practice, Osiris is implemented in Python and per se is dependent on several libraries, which may again conflict with the libraries recommended by *SnifferDog*.

messages. While, in principle, *DockerizeMe* could also be applied to fully analyzed Jupyter notebooks, there are important conceptual differences compared to ours. First, *DockerizeMe* only considers the latest version of packages. It hence cannot deal with programs containing deprecated, removed, renamed APIs in the latest version of packages. Indeed, as empirically reported by Horton et al. [20] by an empirical study about the executability of Python code snippets on GitHub. Their experimental results show that most gists are not executable in a default Python environment, and while a naive approach can infer dependencies for some gists, it fails to do so in the majority of cases. Second, *DockerizeMe* does not *execute* code to validate its findings, let alone compare results against published results—whereas *SnifferDog* automatically determines the configuration that makes the notebook executable, ideally even reproducing the results. The authors further proposed the tool V2 that takes the program crashes information to guide the search for correct environment dependencies [21]. However this approach relies on repeated execution of code snippets and does not handle the case when no crash happen and dependencies are incorrect. On the contrary, *SnifferDog* is a static approach that analyzes dependencies using pre-built knowledge.

B. Studies on Jupyter Notebooks

Despite their popularity, research on Jupyter notebooks is still limited. In 2019, Pimentel et al. conducted a large scale study on the executability and reproducibility issues of over one million selected notebooks [1]. Their experimental results show that around 25% of the notebooks can be executed without any runtime errors, and among which only 4.03% of them can eventually produce the original results. Loenzen et al. [22] empirically investigate the code duplication and reuse in Jupyter notebooks and find that notebook repositories have a mean self-duplication rate of 7.6%. More recently, Wang et al [23] conduct a large scale study on the code quality and empirically find that even notable Jupyter notebooks are frequently suffered from technical debts (e.g., deprecated API uses).

Following the discovery of low-reproducibility issues among Jupyter notebooks, Wang et al. [2] went further to propose to address the root causes leading to non-reproducible notebooks by offering the community a tool called Osiris. This tool attempts to reproduce Jupyter notebooks by leveraging code instrumentation to find out and address the uncertainties when executing Jupyter notebooks. However, due to a lack of appropriate execution environments, around 80% selected notebooks failed to be fully executed. Following this research line, Fangohr et al. [24] have further proposed another tool called *nbval* (implemented as a plugin for pytest) aiming at supporting automated testing and validation of Jupyter notebooks. As argued by the authors, nbval could be leveraged to promote reproducible science such as checking that deployed software behaves as its documentation suggests.

In addition to researches from the Software Engineering community, Jupyter notebooks have been selected frequently as subjects by our fellow researchers in other domains [25]–[33]. For example, Perkel et al. [3] have studied why Jupyter notebooks are popular among data scientists. Kery et al. [26] have introduced a tool named *Verdant* to support users with efficient retrieval and sensemaking of messy version data. It allows users to compare, replay, and trace the relationships amongst different versions of artifacts of both non-code and code in the editors. Furthermore, Rule et al. [34] look into the notebooks from the aspects of human factors, and empirically observed that computational notebooks may lack the explanatory textual information.

C. Dependency Analysis

One of the most representative ones targeting Python dependencies would be the work recently proposed by Ying et al., who attempt to resolve dependency conflicts in the Python library Ecosystem [35]. They designed and implemented a tool named *Watchman* to detect dependency conflicts among libraries indexed by the PyPI repository. They also reported 117 potential dependency issues to the developers of the corresponding projects. Despite Python has become one of the most popular programming languages nowadays, studies on Python projects focus on library API issues and their evolution patterns [36], [37], there has not been much relevant research aiming at resolving Python dependencies. Nevertheless, dependency analysis has been a hot research topic for many other programming languages [38]–[44]. The concepts of these approaches, such as resolving compatibility issues caused by the evolution of libraries [45], [46], automated replacing outdated libraries [47], or updating deprecated library APIs [48], we believe, should also be appliable to Python software applications.

VII. Conclusion and Future Work

Jupyter notebooks may be touted as a prime means to obtain reproducible and replicable research results. In practice, however, they suffer from problems that Software Engineering has solved long ago: bad code quality, insufficient documentation, and—as shown in this paper—little to nonexistent management of dependencies. It will take time until the community of Jupyter notebook authors will learn to see their notebooks not only as entities to be published, but also as living code that should be designed to be readable, reusable, and maintainable. Until then, it will be up to the Software Engineering community to reverse engineer notebooks such that they can be executed and tested.

In this paper, we have taken a major step towards this goal, namely restoring the execution environments of Jupyter notebooks. We found that dependencies are hardly ever stated explicitly, and that this problem seriously impedes re-execution of Jupyter notebooks. By analyzing imports and API usages in notebooks and matching them against Python libraries in various versions, our tool *SnifferDog* can identify library candidates that were used for notebook creation. By searching for candidate configurations that make the notebook executable again (and hence fully reproducible), *SnifferDog*

provides notebook users with essential information that makes notebooks usable again. Given the popularity of notebooks, *SnifferDog* thus shows how Software Engineering can make an important contribution towards reproducible and extensible science.

There is still lots to do, though. Our future work will focus on the following topics:

Larger API Bank. Our API bank is only built based on 488 libraries. While these make up the most popular libraries, having a larger API Bank will further extend the capability of our approach.

Python and C code. A small number of "Cython" libraries combine both C and Python syntax to achieve C-like performances, letting a small set of library APIs be overlooked.

Advanced features. As already stated in Section V, some advanced Jupyter notebook and Python features are not yet supported by *SnifferDog*, notably magic functions and indirect dependencies.

Beyond Python. The *SnifferDog* principles are not limited to Python. We plan to extend *SnifferDog* to other popular Jupyter notebook languages such as R and Julia.

Beyond notebooks. The *SnifferDog* principles also extend beyond notebooks. *SnifferDog* could be equally applied to C source code to determine which library versions would be required for construction and execution. A version of *SnifferDog* for LaTeX that automatically determines required packages and versions may be especially welcome in scientific communities.

SnifferDog is available as open source (with explicit dependencies, of course). A complete *replication package* including all experimental data is available at

https://github.com/SMAT-Lab/SnifferDog.git

References

[1] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of Jupyter notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 507–517.

[2] J. Wang, T.-Y. Kuo, L. Li, and A. Zeller, "Assessing and restoring reproducibility of Jupyter notebooks," in *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*, 2020.

[3] J. M. Perkel, "Why Jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.

[4] E. Horton and C. Parnin, "DockerizeMe: Automatic inference of environment dependencies for Python code snippets," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 328—-338. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00047

[5] "Pip user guide." [Online]. Available: https://pip.pypa.io/en/stable/user_guide/#requirements-files

[6] "Anaconda software distribution," 2020. [Online]. Available: https://docs.anaconda.com/

[7] "The Python language reference." [Online]. Available: https://docs.python.org/3/reference/import.html

[8] "Python module index." [Online]. Available: https://docs.python.org/2.7/py-modindex.html

[9] "Python module index." [Online]. Available: https://docs.python.org/3.1/modindex.html

[10] "Python module index." [Online]. Available: https://docs.python.org/3.2/py-modindex.html

[11] "Python module index." [Online]. Available: https://docs.python.org/3.3/py-modindex.html

[12] "Python module index." [Online]. Available: https://docs.python.org/3.4/py-modindex.html

[13] "Python module index." [Online]. Available: https://docs.python.org/3.5/py-modindex.html

[14] "Python module index." [Online]. Available: https://docs.python.org/3.6/py-modindex.html

[15] "Python module index." [Online]. Available: https://docs.python.org/3.7/py-modindex.html

[16] "Python module index." [Online]. Available: https://docs.python.org/3/py-modindex.html

[17] "Sample size calculator." [Online]. Available: https://www.surveysystem.com/sscalc.htm

[18] "Documentation¶." [Online]. Available: https://setuptools.readthedocs.io/en/latest/

[19] "Built-in magic commands¶." [Online]. Available: https://ipython.readthedocs.io/en/stable/interactive/magics.html

[20] E. Horton and C. Parnin, "Gistable: Evaluating the executability of python code snippets on github," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 217–227.

[21] ——, "V2: Fast detection of configuration drift in python," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 477–488. [Online]. Available: https://doi.org/10.1109/ASE.2019.00052

[22] A. Koenzen, N. Ernst, and M.-A. Storey, "Code duplication and reuse in jupyter notebooks," *arXiv preprint arXiv:2005.13709*, 2020.

[23] J. Wang, L. Li, and A. Zeller, "Better code, better sharing: On the need of analyzing Jupyter notebooks," in *The 42nd International Conference on Software Engineering, NIER Track (ICSE 2020)*, 2020.

[24] H. Fangohr, V. Fauske, T. Kluyver, M. Albert, O. Laslett, D. Cortés-Ortuño, M. Beg, and M. Ragan-Kelly, "Testing with jupyter notebooks: Notebook validation (nbval) plug-in for pytest," *arXiv preprint arXiv:2001.04808*, 2020.

[25] D. Koop and J. Patel, "Dataflow notebooks: encoding and tracking dependencies of cells," in *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.

[26] M. B. Kery and B. A. Myers, "Interactions for untangling messy history in a computational notebook," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 147–155.

[27] M. S. Rehman, "Towards understanding data analysis workflows using a large notebook corpus," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1841–1843.

[28] A. Rule, I. Drosos, A. Tabard, and J. D. Hollan, "Aiding collaborative reuse of computational notebooks with annotated cell folding," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–12, 2018.

[29] S. Samuel and B. König-Ries, "ProvBook: Provenance-based semantic enrichment of interactive notebooks for reproducibility." in *International Semantic Web Conference (P&D/Industry/BlueSky)*, 2018.

[30] A. Watson, S. Bateman, and S. Ray, "PySnippet: Accelerating exploratory data analysis in Jupyter notebook through facilitated access to example code," in *EDBT/ICDT Workshops*, 2019.

[31] H. Nguyen, D. A. Case, and A. S. Rose, "NGLview–interactive molecular graphics for Jupyter notebooks," *Bioinformatics*, vol. 34, no. 7, pp. 1241–1242, 2018.

[32] H. Fangohr, M. Beg, M. Bergemann, V. Bondar, S. Brockhauser, C. Carinan, R. Costa, C. Fortmann, D. F. Marsa, G. Giovanetti *et al.*, "Data exploration and analysis with jupyter notebooks," in *17th Biennial International Conference on Accelerator and Large Experimental Physics Control Systems*, no. TALK-2020-009, 2019.

[33] M. García-Domínguez, C. Domínguez, J. Heras, E. Mata, and V. Pascual, "Jupyter notebooks for simplifying transfer learning," in *International Conference on Computer Aided Systems Theory*. Springer, 2019, pp. 215–221.

[34] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.

[35] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: monitoring dependency conflicts for python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.

[36] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated Python library APIs are (not) handled," in *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[37] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do Python framework APIs evolve? an exploratory study," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 81–92.

[38] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "Opium: Optimal package install/uninstall manager," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 178–188.

[39] J. Patra, P. N. Dixit, and M. Pradel, "ConflictJS: finding and understanding conflicts between JavaScript libraries," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 741–751.

[40] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in Maven Central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 333–343.

[41] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 319–330.

[42] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung, "Could I have a stack trace to examine the dependency conflict issue?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 572–583.

[43] L. Li, T. Riom, T. F. Bissyandé, H. Wang, J. Klein, and Y. Le Traon, "Revisiting the impact of common libraries for android-related investigations," *Journal of Systems and Software (JSS)*, 2019.

[44] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*, 2020.

[45] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[46] H. Cai, Z. Zhang, L. Li, and X. Fu, "A large-scale study of application incompatibilities in android," in *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*, 2019.

[47] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 35–45.

[48] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Cda: Characterising deprecated android apis," *Empirical Software Engineering (EMSE)*, 2020.

Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs

Chenguang Zhu[1], Ripon K. Saha[2], Mukul R. Prasad[2], and Sarfraz Khurshid[1]
[1]The University of Texas at Austin, [2]Fujitsu Research of America, Inc.
Email: cgzhu@utexas.edu, rsaha@fujitsu.com, mukul@fujitsu.com, khurshid@utexas.edu

2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678889

***Abstract*—Data scientists typically practice exploratory programming using computational notebooks, to comprehend new data and extract insights. To do this they iteratively refine their code, actively trying to re-use and re-purpose solutions created by other data scientists, in real time. However, recent studies have shown that a vast majority of publicly available notebooks cannot be executed out of the box. One of the prominent reasons is the deprecation of data science APIs used in such notebooks, due to the rapid evolution of data science libraries. In this work we propose RELANCER, an automatic technique that restores the executability of broken Jupyter Notebooks, in near real time, by upgrading deprecated APIs. RELANCER employs an iterative runtime-error-driven approach to identify and fix one API issue at a time. This is supported by a machine-learned model which uses the runtime error message to predict the kind of API repair needed - an update in the API or package name, a parameter, or a parameter value. Then RELANCER creates a search space of candidate repairs by combining knowledge from API migration examples on GitHub as well as the API documentation and employs a second machine-learned model to rank this space of candidate mappings. An evaluation of RELANCER on a curated dataset of 255 un-executable Jupyter Notebooks from Kaggle shows that RELANCER can successfully restore the executability of 56% of the subjects, while baselines relying on just GitHub examples and just API documentation can only fix 38% and 36% of the subjects respectively. Further, pursuant to its real-time use case, RELANCER can restore execution to 49% of subjects, within a 5 minute time limit, while a baseline lacking its machine learning models can only fix 24%.**

***Index Terms*—data science, API migration, software evolution**

I. INTRODUCTION

The ready availability of sensors and inexpensive compute resources, coupled with a number of significant advances in machine learning and data analytics has fueled an explosive growth in the field of *data science* [1], [2]. Kaggle, the relatively nascent online community of data scientists, already boasts over 6 million users and hosts 50,000 public datasets [3].

Data scientists typically follow the paradigm of *exploratory programming*, iteratively refining code to comprehend new data and extract meaningful insights from it [4], [5]. To do this they actively try to re-use and re-purpose solutions created by other data scientists, in *real time*. Indeed, Kaggle proudly claims that: *"Inside Kaggle you'll find all the code & data you need to do your data science work"* [3]. Jupyter Notebooks [6] - interactive computational structures which interleave code snippets, natural language text, computation results, and visualizations - have become the medium of choice for data scientists to program, record, and *share* data analyses [7], [8]. Kaggle hosts 400,000 such public notebooks [3]. Similarly, there are currently nearly 10 million Jupyter Notebooks hosted on GitHub, with the number having grown 8-fold over just the last two years [9].

A data scientist pursuing exploratory programming by re-using existing Jupyter Notebooks would require such notebooks to be easily *reproducible* or at the least *executable*. In fact, producing *"reproducible computational workflows"* was intended to be one of the defining features of Jupyter Notebooks (and computational notebooks in general) [10], [11]. However, this promise has not been realized in practice. A large-scale empirical study by Pimental et al. on Jupyter Notebooks projects on GitHub found that only 24.11% of their selected notebooks could execute without errors, and only 4.03% reproduced the original results [12]. The concerns about executability are echoed in a recent survey of data scientists [13]. Similarly, our study of a sample of machine learning notebooks on Kaggle (reported in Section II-A) revealed that 47% of them could not be executed.

Recent studies have identified a number of root causes for why archived Jupyter Notebooks cannot be easily re-executed. These include, ambiguity in the execution order of the notebook cells [14], lack of knowledge of the notebook's execution environment [15], and references to external resources (such as data on external servers) [12], [14]. An emerging body of work also aims at restoring the executability of such notebooks by automatically inferring correct cell execution order [14] or inferring an appropriate execution environment [15]. Other studies have highlighted the landscape of rapidly evolving Python libraries, particularly data science libraries, which leads to unhandled deprecation issues in programs using these libraries [16], [17], [18]. In fact, our study of Kaggle notebooks (Section II-A) found that deprecated APIs was at least one of the reasons (if not the only reason) for the unexecutability of at least 31% of the unexecutable notebooks. Therefore, in this paper, we propose a technique to restore the executability of Jupyter Notebooks by automatically diagnosing, inferring, and upgrading deprecated APIs used by the notebook.

There is a rich body of existing work on computing API mappings across pairs of libraries, languages, or platforms, to be then used to migrate a project from a source library (platform) to a target library (platform) [19], [20], [21], [22], [23], [24], [25], [26]. There is also recent work that can perform an end-to-end migration [27], [28], [18]. However, our target use-case - automatically making broken Jupyter Notebooks executable in near real-time, by fixing deprecated

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00031

APIs - presents some unique challenges and opportunities, which precludes the direct application of existing solutions. First, many of the API mapping techniques [20], [23], [29], [30], [31], [32] only create the mappings rather than perform an end-to-end migration, which our use-case mandates. Second, all the techniques that perform complete migration [27], [28], [18] rely on a reference implementation in the source library/platform as a strong oracle for migration. Our use-case lacks such a reference implementation - we only have an unexecutable notebook. Third, each of the existing techniques rely on a single information source to compute the mappings or migration, typically either existing examples of updates (e.g., on GitHub) [27], [28] or API documentation [18]. However, given the rapid evolution of data science libraries, public examples of recent API changes may not exist [18]. Further, even the API documentation may not comprehensively capture all deprecation information [16]. This motivates a migration technique that combines knowledge from both sources. Finally, our use-case of supporting exploratory programming requires a technique that can fix broken notebooks (chosen by the user for exploration) in *near real time*. Traditional API migration techniques are not similarly constrained.

Insights. We design our approach based on three key insights. First, we observe that individual API upgrades (even different instances of the same change) can be successfully diagnosed and performed independent of each other, i.e., one instance at a time. Second, we build on the observation made in other recent work [17], [18] that runtime error messages in Python, in particular those of Python data science programs, are often accurate enough to unambiguously diagnose the broad cause of error. Specifically, we observe that for deprecated APIs the runtime error can be used to accurately flag the *kind* of program element requiring a change - the API name or package name, a parameter, or the value of a parameter. The third insight is that previous instances of API upgrades (e.g., on GitHub) and API library documentation are complementary sources of knowledge that can collectively inform an automated API migration technique.

Proposed approach. Based on these insights, in this work we propose RELANCER [1], a technique for automatically restoring the executability of broken Jupyter Notebooks, in near real-time, by upgrading deprecated APIs. RELANCER employs a divide-and-conquer approach to upgrade a broken notebook, using runtime errors to iteratively identify and fix one API issue at a time. This iterative strategy is supported by two key components. The first component is a machine learned model that uses the runtime error message to predict the *kind* of repair action to perform, specifically a change in the API name (or package), a parameter name, or a parameter value. This decision accelerates the search by directing focus to the appropriate search space. RELANCER creates this space by aggregating candidates from both GitHub examples as well as API documentation. In the case of name changes to the API or its package, this space could be fairly large. Thus, RELANCER uses a second machine learned model to produce a ranked list of candidate API mappings organically combining knowledge from the two sources.

We evaluate RELANCER on a curated dataset of 255 unexecutable Jupyter Notebooks from Kaggle. RELANCER can successfully restore the executability of 56% of the subjects, while baselines relying on just GitHub examples and just API documentation can only fix 38% and 36% of the subjects respectively. We note that a response time of up to a few minutes is recognized as real-time or near real-time performance for Big Data and data analytics applications [33], [34]. Therefore, we use 5 minutes as a threshold of near real-time performance for the purposes of this paper. Pursuant to its real-time use case, within a 5 minute time-span RELANCER can restore execution to 49% of subjects while a baseline lacking its machine learning models can only fix 24%.

This paper makes the following contributions:

- **Problem:** We highlight the problem of restoring execution of Jupyter Notebooks by upgrading deprecated APIs, in real time, in order to support the use-case of exploratory programming typically employed by data scientists.
- **Technique:** An automated machine learning based technique RELANCER, to orchestrate this upgrade for unexecutable notebooks.
- **Evaluation:** An evaluation of RELANCER on a curated dataset of 255 unexecutable Jupyter Notebooks mined from Kaggle, and ablation studies comparing its performance against 5 different baselines.
- **Implementation & Dataset:** A public release of the source code and dataset used in this paper, to promote replication and open science, is available at https://sites.google.com/view/relancer.

II. MOTIVATION

In this work we restrict our scope to Jupyter Notebooks on Kaggle, the most popular online forum for data scientists. Further, we focus on notebooks written in Python, the language of choice in data science [9] and specifically ones performing predictive (i.e., machine learning (ML)) tasks, since ML is one of most important end-points for data science.

A. A Study on Kaggle Notebooks

We hypothesize that the problem of unexecutable notebooks, reported by recent studies [12], [14], [15] for GitHub projects, is also mirrored on Kaggle. We further hypothesize that the issue of deprecated DS APIs, due to the rapid evolution of DS libraries [16], [18] should be a significant factor in the unexecutability. To validate these hypotheses, we conduct a lightweight study of Jupyter Notebooks on Kaggle.

Data Collection. We use Meta Kaggle [35], the official dump of Kaggle meta data, as the data source for this study. Meta Kaggle links to Jupyter Notebooks on Kaggle, including competitions, datasets, kernels, and discussions on a daily basis. We downloaded the snapshot of Meta Kaggle on July 20,

[1] from the French word meaning *to revive*, since RELANCER revives broken notebooks by restoring their executability.

2020 for this study. It consists of 49,061 datasets and 381,556 notebooks (kernels).

Sampling. To keep the study simple and efficient, we impose several filtering criteria. Starting with Python notebooks for predictive tasks, we exclude datasets containing media data (e.g., images, audio, and video), or spanning over multiple or large files (i.e., >500 MB). We calculated that we require a sample size of 4,000 notebooks to achieve a 99% confidence level within a margin of error of $\pm 2\%$ [36]. To this end, using upvotes on Kaggle as a proxy of quality, we sorted all datasets that passed our filtering criteria by the number of upvotes and similarly notebooks for each dataset by the same metric. We selected top 100 notebooks (or as many available if fewer) per dataset, to respect diversity. This yielded 4043 notebooks from the top 350 datasets.

Execution Environment. To execute the Jupyter Notebooks, we convert them into Python programs using `nbconvert` [37]. We set up a unified virtual execution environment on top of the standard Anaconda [38] scientific computing environment of Python 3.6, also used in other studies [12], [14], [15]. This standard environment includes all the popular libraries (over 200 library packages). Further, we performed a static import analysis on our notebook corpus and manually installed the 20 most commonly used Python libraries.

Execution & Analysis. Among the 4043 notebooks in our study corpus, 2,155 notebooks executed successfully while 1,888 notebooks, i.e., 47% of the total corpus did not execute due to at least one error. Consistent with previous studies on GitHub [12], [14], this shows that unexecutable notebooks is also a significant problem on Kaggle, although the exact numbers differ due to platform differences.

Deprecation Errors. To quantify the impact of API deprecation on notebook executability, we created a ground-truth list of deprecated APIs by manually inspecting the release notes of 12 popular DS libraries, namely: scikit-learn [39], pandas [40], seaborn [41], NumPy [42], SciPy [43], XGBoost [44], Plotly [45], TensorFlow [46], Keras [47], statsmodels [48], imbalanced-learn [49], and CatBoost [50]. These libraries are a subset of the top-20 libraries found in our Kaggle study, which systematically report API deprecations in their documentation. In total, we manually identified 317 upgrades (i.e., deprecations) through the release notes, including 198 function name deprecations and 119 parameter deprecations. Checking these against our corpus of 1,888 unexecutable notebooks showed that 582 of the notebooks, i.e., 31% contained at least one of these deprecated APIs.

B. A Motivating Example

Figure 1 presents a fragment of a real-world Jupyter Notebook on Kaggle [51] with a patch generated by RELANCER, that fixes all API deprecation errors. The original notebook, submitted in 2017, is a high-quality notebook that won a bronze medal on Kaggle. Further, it has gathered 37,767 views and 161 forks (as of April 12th, 2021). This indicates that the notebook has been very useful to other data scientists for learning or expediting their work. However, the original notebook no longer executes with the latest default Anaconda environment due to several deprecated APIs. In particular, Figure 1 shows five locations containing three deprecated APIs:

1) `sklearn.cross_validation.train_test_split`
2) `sklearn.cross_validation.grid_search.GridSearchCV`
3) `sklearn.cross_validation.ShuffleSplit`

The first two APIs are deprecated due to a change in the package structure, i.e., moving from `sklearn.cross_validation` to `sklearn.model_selection` [52]. This kind of deprecation is called *function deprecation* [16]. The deprecation in third API is more complicated. In addition to the aforementioned package change, the parameter `n_iter` was replaced by `n_splits` starting from version 0.18 [52]. This is called *parameter deprecation* [16]. Therefore, if a data scientist wants to run the notebook on her own machine, often the first step of trying to re-use it, she needs to laboriously find, diagnose, and fix each of the deprecation errors, one by one. This is a big barrier to re-use an otherwise high-quality notebook.

Although API migration is a well-established research area and there are even a few recent techniques that perform end-to-end API migration [27], [28], [18], the Jupyter Notebook in Figure 1 would be outside the scope of these techniques. First, these techniques [27], [28] rely on a single information source, typically either GitHub or the API documentation. However, for the aforementioned example, the patch for `ShuffleSplit` with the argument change is not directly available on GitHub. On the other hand, a tool that only uses documentation may not find the correct mapping for other deprecated APIs in the example easily since `cross_validation` and `model_selection` are not textually similar. A brute-force trial of each API in the documentation would be impractical, especially with multiple deprecated APIs. Specifically, in our experiment, this notebook could not be fixed using only documentation within a 30-minute time limit (Section V, RQ2).

To overcome these challenges, RELANCER effectively combines two primary sources of information - GitHub examples and API documentation - and fixes the problem iteratively guided by the error message. Specifically, RELANCER first executes the original notebook, and find an `ImportError` at line 235, as shown in Figure 2.

At this point, the repair action predictor component of RELANCER automatically identifies that the fully qualified name of a module needs to be changed. RELANCER identifies the affected module name from the error message and also identifies and localizes the affected API by analyzing the abstract syntax tree (AST) of line 240. Then RELANCER searches for potential mappings: $API_{old} \rightarrow API_{new}$ on GitHub and documentation and passes them to its learning-to-rank model. For this specific case, RELANCER ranks the correct API_{new}: `model_selection.train_test_split` on top. The reason is that although the textual similarity between old and new package names is low, this change is frequent on Github. While applying a patch, RELANCER not only fixes the actual reference

```
1  # kaggle.com/sagarnildass/predicting-boston-house-prices
... ...
22 import numpy as np # linear algebra
... ...
235 -from sklearn import cross_validation
235 +from sklearn import model_selection
... ...
240 -X_train, X_test, y_train, y_test = cross_validation.train_test_split(features, prices, test_size=0.2, random_state=42)
240 +X_train, X_test, y_train, y_test = model_selection.train_test_split(features, prices, test_size=0.2, random_state=42)
241  print("Training and testing split was successful.")
... ...
284  from sklearn.tree import DecisionTreeRegressor
285 -from sklearn.cross_validation import ShuffleSplit
285 +from sklearn.model_selection import ShuffleSplit
... ...
290  # Create 10 cross-validation sets for training and testing
291 -cv = ShuffleSplit(X.shape[0], n_iter=10, test_size=0.2,  random_state=0)
291 +cv = ShuffleSplit(X.shape[0], n_splits=10, test_size=0.2,  random_state=0)
292  train_sizes = np.rint(np.linspace(1, X.shape[0]*0.8-1,9)).astype(int)
... ...
514  from sklearn.metrics import make_scorer
515 -from sklearn.grid_search import GridSearchCV
515 +from sklearn.model_selection import GridSearchCV
516 ...
```

Fig. 1: A motivating and illustrative example of RELANCER.

```
Traceback (most recent call last):
File "predicting-boston-house-prices.py", line 235, in <module>
from sklearn import cross_validation
ImportError: cannot import name 'cross_validation'
```

Fig. 2: The error message thrown by line 235.

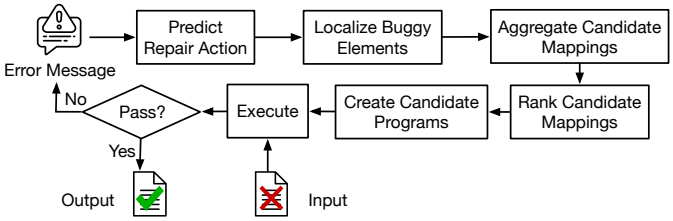

Fig. 3: Overview of RELANCER.

but also fixes the necessary import statements. For example, as a result of changing `cross_validation.train_test_split` to `cross_validation.train_test_split`. RELANCER also fixes lines 235 and 285. Now after validation, RELANCER finds a new error message pointing to another error at line 291.

Although the deprecated API involved two issues: deprecated module name and parameter name, the first issue at line 285 is already fixed by RELANCER while fixing the previous API. Therefore, the new error shows a `TypeError` at line 291. RELANCER predicts that the appropriate repair action for this error is a parameter name change and identifies the problematic parameter name from the error message. Then RELANCER again leverages Github and documentation to get all the candidate parameter names and validates one by one. In this way, RELANCER fixes all the deprecated errors until the notebook fully executes.

In summary, this is a large notebook that contains 701 lines of code containing five unique API deprecation errors at seven locations. To give an idea about the number of candidate patches, RELANCER had on average 355 choices per location. However, RELANCER systematically narrowed down the search space and fixed all the errors in ten minutes.

III. RELANCER

Given a Jupyter Notebook that does not currently execute, due to one or more deprecated APIs, RELANCER automatically applies necessary program transformations to upgrade all the deprecated APIs so that the resulting notebook executes without any error. Figure 3 presents an overview of RELANCER. At a high level, RELANCER iteratively performs the following operations until the notebook is error-free:

1) **Analysis of Error.** This step analyzes the current error message: first to predict the atomic repair action that is required to solve that particular error and then to identify the deprecated API.
2) **Aggregation of Candidate Mappings.** This step mines the Github repositories and API documentation to aggregate the candidate mapping between program elements: $PE_{dep} \rightarrow PE_{new}$ where a PE is an API or a parameter name, or an argument value.
3) **Ranking of Candidate Mappings.** This step employs a learning-to-rank model to rank the candidate mapping in such a way that the more promising a mapping is, the higher it is in the ranking.

4) **Creation and Validation of Modified Programs.** This step creates a candidate program for each mapping one by one based on the ranking, and executes it to check if it fixes the current error.

The subsequent sections describe each of the above steps of RELANCER in more detail.

A. Step-1: Analysis of Error Message

The analysis of error messages consists of two parts, namely (1) predicting the repair action to take and (2) localizing the buggy element to apply it on.

1) Prediction of Repair Action: There are mainly three kinds of API deprecation issues in Python programs: *function deprecation*, *parameter deprecation*, and *parameter's value deprecation* [16]. Broadly, these map to the kind of repair action required to fix the bug. We make the observation that typically the error message is indicative of the kind of deprecation. For example, fixing a function deprecation requires renaming the API. Some example errors for such a deprecation are:

ImportError: cannot import name 'jaccard_similarity_score'.

ModuleNotFoundError: No module named 'sklearn.grid_search'.

Intuitively, there are some common patterns in the messages, such as a particular name cannot be imported or found. However, they are not exactly the same and may vary across libraries and APIs. To leverage this information systematically and robustly, we build a machine learning classifier to predict the repair action directly from the error message. Specifically, the classifier predicts one of three different repair actions: i) changing the fully qualified name of a function, ii) changing parameter name, and iii) changing an argument value.

Building the Classifier. We first apply standard text preprocessing techniques to clean the error message, such as removing non-alphabetic characters, tokenizing and normalizing text. Then we convert the text into a matrix of token counts where each word becomes a feature and value represents the count of that token. So after this step, each error message in the training dataset becomes a tuple of numeric values representing text in the error message and the target is the corresponding repair action. To predict the repair action from the error message, we use Linear Support Vector Classification [53] since it is generally accepted that it is well suited for text classification [54].

2) Creation of Training Data: To create a training dataset for the repair action predictor, we need instances of pairs of buggy and fixed versions of notebooks corresponding to deprecated APIs, with the buggy version providing the error message and an executable fixed version as the ground truth for the required repair action. However, Kaggle does not provide for the tracked versioning and collaborative project editing (like GitHub) to facilitate gathering such instances. Therefore, we devise a novel mutation-based technique to systematically create our training data.

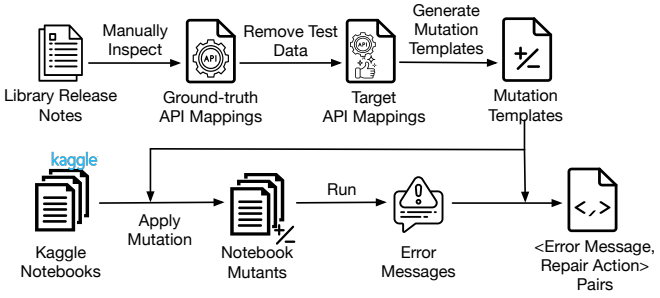

Fig. 4: Mutation-based training dataset creation.

Approach. Our approach is inspired by techniques for data augmentation [55] and simulation-based training data generation [56], [57], [58] used to generate synthetic, but realistic, training data for machine learning. Specifically, we use the insight that given a known instance of an API deprecation $PE^i_{dep} \rightarrow PE^i_{new}$ (which by definition states the repair action) and a working notebook that uses the API PE^i_{new}, we can re-create the buggy version of the notebook by the deprecation mutation $PE^i_{new} \rightarrow PE^i_{dep}$, and execute the buggy notebook to generate the error message. This provides a pair of an error message and its corresponding repair action, i.e., a single instance of training data. Similarly, we use the set of manually collected deprecation mapping $\{PE^1_{dep} \rightarrow PE^1_{new}, PE^2_{dep} \rightarrow PE^2_{new}, ...\}$ and the set of executable notebooks identified in our Kaggle study (Section II-A), and systematically inject deprecation mutations into executable notebooks that are using the new version of the respective API. Figure 4 summarizes our overall workflow.

3) Localization of Buggy Program Elements: As Figure 2 shows, typically an error message describes a single problem and consists of a stack trace, followed by an error description in natural language text. RELANCER exploits this broad structure of the error message to extract (1) the buggy line number in the main file and (2) the specific program element on it, implicated in the deprecation. To do this, RELANCER analyzes the error message along with the corresponding source file as follows.

RELANCER first extracts the line number of the most recent (last) call site for the main source file from the stack trace in the error message–e.g., Line 235 in Figure 2. Second, it extracts all the deprecation-related program elements, specifically module/function/parameter names and values at that line from the AST of that source file. Third, RELANCER searches all the candidate program elements that it identified from the source code, in the text portion of the error message to pinpoint the exact program element that the error is complaining about. For our motivating example, it is the `cross_validation` module.

B. Step-2: Aggregation of Candidate Mappings

Given a deprecated program element PE_{dep} identified in the previous step, where PE is a module name, a function name, a parameter name, or a parameter value, RELANCER utilizes two sources of information: i) the API documentation

and ii) Github to identify all possible candidate mappings: $PE_{dep} \rightarrow PE_{new}$.

1) Mining API documentation: The objective of this step is to create a knowledge base offline that consists of each API from the latest version of each library under consideration. Most of the popular libraries use a very similar format document structure for their API documentation. For each API method, RELANCER parses (using *Beautiful Soap* [59]) the HTML page to collect its parameter names and the set of possible discrete values (if available). It then stores the information of these libraries in an internal JSON database.

2) Mining of Github: Since RELANCER focuses on fixing one PE_{dep} at a time, its on-demand GitHub search identifies a list of potential PE_{new} that can replace PE_{dep}.

Identifying API Upgrade Mappings. To identify plausible API upgrades for a given previously unseen API, we leverage the idea of mining code changes[60]. RELANCER leverages Github's own search engine through the provided REST API [61] to search relevant example changes efficiently.

First, given a deprecated API, F_{dep} that has been identified in the previous step and needs to be fixed, RELANCER tokenizes the fully qualified name of F_{dep} and then constructs a GitHub search query by taking each token *plus* the keywords: "update", "upgrade", "replace", and "deprecate". Then RELANCER executes this query on GitHub using GitHub's REST API, which returns a list of commits sorted by GitHub's "Best Match" metric [62] in descending order.

Second, RELANCER excludes irrelevant commits, i.e., commits that do not contain an upgrade of F_{dep}. Such irrelevant commits can be introduced due to GitHub's imprecision of searching. For each commit in searching result, RELANCER extracts the abstract syntax tree (AST) of the program version before (V_0) and after (V_1) the commit. It discards a commit if the AST of V_0 does not include F_{dep}.

Third, RELANCER analyzes the AST difference of V_0 and V_1. For a deleted deprecated API call node F_{dep} in V_0, we consider an added node in V_1 as an update of F_{dep} if the deletion and addition occur at the same level of the AST, and the deleted and added nodes are under the same parent. If there are multiple addition candidates for a deletion, we pair the deleted node with the addition candidate whose name has the highest textual similarity with it. After AST node changes are identified, RELANCER discards any commit that does not involve a change in F_{dep}. Finally, for each update of F_{dep}, RELANCER extracts the corresponding new API: F_{new}, adding them to a list. This resulting list consists of the candidate APIs mined from GitHub and is cached (stored locally) for future use, to avoid repeating the search if RELANCER encounters the same deprecated API call multiple times.

Identifying Parameter Mappings. The APIs in data science libraries generally have a large set of parameters but few discrete options per parameter [63]. Furthermore, popular data science programming languages like Python can accept parameters in the form of key-value pairs. Therefore, mining the exact change (patch) that RELANCER needs for the subject error is challenging. However, since RELANCER narrows down the scope of search in the previous steps quite a lot: i) by first fixing function deprecation by F_{new} and ii) then identifying the deprecated parameter name P_{dep} from the error message, it launches a focused search to collect the set of all parameter names and the values developers used on GitHub for that specific API. Finally, RELANCER returns a map where the keys are comprised of all the parameter names and values represent the set of values that parameter can take. Then a parameter mapping: $P_{dep} \rightarrow P_{new}$ is established where P_{new} is not currently used in the current call site.

3) Compiling the Final List of Candidate Mappings: As we discussed in Section III-A, RELANCER allows three kinds of upgrades: i) API name changes, ii) parameter name changes, and iii) argument value changes. Furthermore, RELANCER performs one single upgrade at a time. Therefore, for a given deprecated program element, i.e., an API, a parameter, or a value, RELANCER takes a union of both sets of elements mined from: i) the API documentation and ii) GitHub to compile a final set of candidates: $PE_{dep} \rightarrow \{PE^1_{new}, PE^2_{new}, .., PE^n_{new}\}$.

C. Step-3: Ranking of Candidate Mappings

For a given deprecated API: F_{dep}, there can be hundreds of candidate APIs: $\{PE^1_{new}, PE^2_{new}, .., PE^n_{new}\}$. Further, there can be multiple errors in a program, which can increase the search space exponentially. Therefore, an effective ranking strategy is important for the success of RELANCER. Learning to rank is an effective ranking strategy that has been found useful in many applications, including various software engineering tasks such as defect prediction [64] and bug localization [65]. Inspired by these applications, we develop a machine learning based learning-to-rank strategy to rank all the candidate fixes for a given deprecated API. However, for any machine learning technique, designing the features that are related to the target is critical. We design the following four features for our task: two features from the API documentation and two features from GitHub.

- **Occurrences on GitHub (OG).** The number of times RELANCER found a mapping: $F_{dep} \rightarrow F_{new}$ on GitHub commits.
- **Percentage of OG (POG).** It is calculated by the ratio of OG to the number of times RELANCER found a mapping where F_{dep} exists.
- **Distance between Fully Qualified Names (D_{FQN}).** RELANCER computes the distance between the fully qualified name of F_{dep} and F_{new} by the Damerau–Levenshtein Distance algorithm [66]. Then the value is normalized between 0 and 1, with zero corresponding to identical names.
- **Distanceof Simple Name (D_{Simple}).** RELANCER follows the same approach as D_{FQN}, but it computes the scores between the simple names (without considering package or module name) of F_{dep} and F_{new}.

Training Learning-to-Rank Model. We create training data from our ground-truth dataset that are not part of test data. For each ground-truth mapping: $F_{dep} \rightarrow F_{new}$, F_{new} is used as a *positive instance* in the training dataset. However, to learn the

characteristics of meta-features for the correct mappings, we also need some *negative candidate mappings* that syntactically represent a correct replacement but functionally not. So we use the same technique described in Section III-B to find a complete list of candidate program elements and randomly choose four negative examples for F_{dep}. We follow the same procedure to collect all the positive and negative instances for each target mapping in the ground truth dataset. Then we compute all the four aforementioned features for each of the positive and negative mappings. Therefore, our final training dataset has a collection of data samples where each row presents an old to new API mapping in terms of the four feature values along with the information whether it is a valid mapping.

Now we formulate our learning-to-rank algorithm as a binary classification problem, where our objective is to determine whether a particular mapping is correct. To this end, we use LightGBM [67] as our classification model, which is a widely used machine learning technique in practice.

Ranking Candidates in Operation. In operation, for a given deprecated API: F_{dep}, RELANCER gets all the candidates: $\{F^1_{new}, F^2_{new}, .., F^n_{new}\}$ from Step-2 and passes to the trained LightGBM model. For each candidate (F^i_{new}), the model computes a probability score for being it to be correct. Then RELANCER uses that probability score to rank all the candidate mappings. It should be noted that once function mapping is found and parameter name or value is identified from the error message, the number of candidates for a particular parameter is not large. Therefore, RELANCER follows the order in the documentation to rank parameter mappings.

D. Step-4: Creation and Validation of Modified Programs.

Given a ranked list of candidate mappings, RELANCER starts iterating through each mapping from the top of the ranked list. For a given mapping: $PE_{dep} \rightarrow PE_{new}$, RELANCER replaces PE_{dep} by PE_{new} to create a new version of the candidate program and validates the change through execution. If the type of error message is unchanged, RELANCER moves to the next mapping. However, if RELANCER gets a new error message, it assumes that it fixed the current error and goes to Step-1 to analyze the error message. RELANCER continues these steps until there is no error in the notebook or the entire search space of candidate mappings is exhausted, or a given time limit is exceeded.

IV. EXPERIMENTAL SETUP

A. Implementation

RELANCER is implemented in the Python programming language. It has mainly three components: i) mining API documentation, ii) on-demand searching of edit-examples on Github, and iii) AST manipulation and validation. We use *Beautiful Soup* [59] for parsing the API documentation HTML pages to extract relevant information. We use *nbconvert* [37] to convert Jupyter Notebooks into Python programs to ease the process of automatic running and generating error messages. We use GitHub*REST API* [61] for searching potential examples on GitHub and then use *LibCST* [68], a popular static analysis framework for Python to analyze AST diff and so on. We also use LibCST for making necessary changes to generate potential fixes for the deprecation errors. We use *scikit-learn* [39] and *LightGBM* [67]–two popular open source machine learning libraries to implement our ML models.

B. Creation of Dataset

Since RELANCER is a machine learning based technique, to evaluate it properly, we need a training dataset and a test dataset that are *mutually exclusive*. Our preliminary study in Section II-A, yielded 582 unexecutable notebooks that contain at least one deprecated API. However, since the objective of RELANCER is to fix only deprecated errors, we wanted to filter out those notebooks that have irrelevant errors. However, it is very challenging to detect all such irrelevant errors.

Error analysis provides an automated way to detect candidate deprecated errors since a previous study [17] suggests that certain kinds of error messages are related to API usage problems, which can lead us to deprecated errors. However, an error message only gives a description of the first error during the execution of the program. Therefore, we targeted capturing at least those notebooks where the first error is related to deprecated APIs. We performed an automated analysis on the error messages to identify the initial list of candidate deprecated errors that contains one of the five errors: i) import error, ii) module not found error, iii) type error, iv) value error, or v) attribute error. Then we searched those APIs in the set of our ground-truth API dataset. Finally, we identified 255 notebooks where the first failure is due to a deprecation error. Therefore, we use all the 255 notebooks as our test dataset to evaluate RELANCER. Table I provides more details about the dataset.

C. Training RELANCER

Training Repair Action Model. To keep the training dataset and test dataset mutually exclusive, our mutation based framework (described in Section III-A2) did not use any deprecated APIs, $F_{test} = \{F^1_{test}, F^2_{test}, ..\}$ from the test dataset. To this end, we eliminated F_{test} from our ground-truth dataset, F_G and only used the remaining deprecated APIs ($F_{tr} = F_G - F_{test}$) to design mutation operators. We use all the 2,155 executable notebooks obtained in our preliminary study (Section II-A) as seed programs to create the training data as described in Section III-A2. Finally, we constructed a training dataset of 500 <error message, repair action> pairs.

We also followed the approach described in Section III-C to construct the training data for the learning-to-rank model from F_{tr}. This gave us a training dataset of 485 instances.

D. Research Questions

We evaluate RELANCER with respect to the following research questions.

RQ-1: How effective is RELANCER at fixing API deprecation issues of Jupyter Notebooks?

RQ-2: Does RELANCER need different sources of information to perform well?

RQ-3: Do the machine learning models in RELANCER help with upgrading more APIs in near real-time?

TABLE I: Subjects: Jupyter Notebooks on Kaggle

Description	Total	Max	Min	Median	Average
#Datasets	110	-	-	-	-
#Notebooks	255	-	-	-	-
#Notebooks per dataset	-	13	1	1	2.32
#Libraries	8	-	-	-	-
LOC	-	739	15	120	152.02
LOC (including comments)	-	1193	21	208	263.81

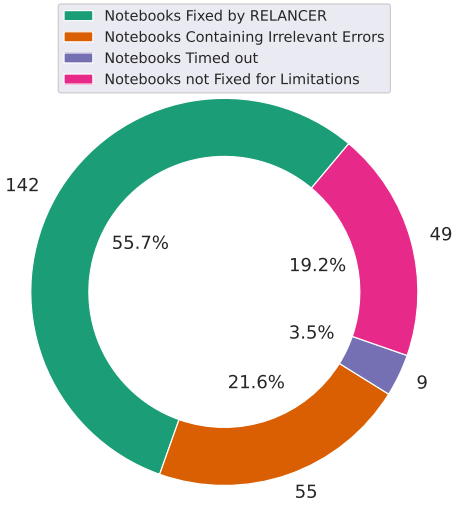

Fig. 5: Effectiveness of RELANCER

E. Experimental Configurations

All the experiments are performed on a 4-core Intel(R) Core(TM) i7-8650 CPU @ 1.90 GHz machine with 16GB of RAM, running Ubuntu 16.04, with Python 3.6.10 and Conda 4.7.10. We set a timeout of 30 minutes for each notebook.

V. EVALUATION

A. RQ-1: Effectiveness of RELANCER

To evaluate the effectiveness of RELANCER, we ran it on all the 255 notebooks in our test dataset with a timeout

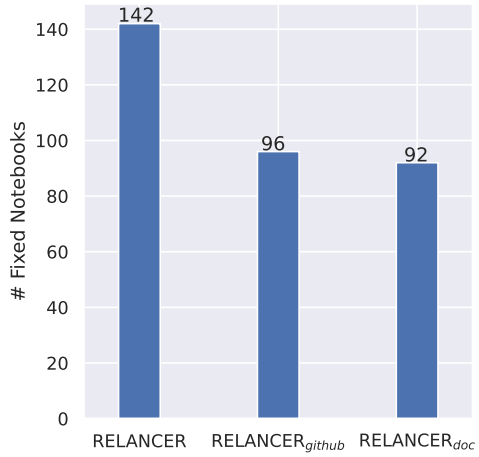

Fig. 6: Effect of removing one source of information

```
-from sklearn.metrics import jaccard_similarity_score
+from sklearn.metrics import jaccard_score
 ...
 knn_yhat = neigh.predict(X_test)
-print("KNN Jaccard index: %.2f" %
 jaccard_similarity_score(y_test, knn_yhat))
+print("KNN Jaccard index: %.2f" %
 jaccard_score(y_test, knn_yhat, average="micro"))
```

Fig. 7: Another illustrative example of RELANCER.

of 30 minutes for each notebook. Then we measure the effectiveness in terms of the number of fixed notebooks and time to fix the notebooks completely. Figure 5 shows that RELANCER was able to fix 142 out of 255 notebooks completely. Among the 142 notebooks it fixed, 107 notebooks required upgrading only the fully qualified name of APIs. Other 35 notebooks required changing parameter names or argument values with or without the fully qualified name of APIs. As we have shown in the motivating example, RELANCER can fix multiple errors in notebooks that require sophisticated fixes in both API's fully qualified name and parameters. The highest number of deprecated APIs RELANCER has upgraded in a single notebook is eight. Further, it should be noted that Python is a rich language that allows for optional parameters which use a default value if that is not explicitly given by developers. However, sometimes even that default value can be deprecated, leaving the notebook broken. In this case, RELANCER can explicitly pass the problematic parameter with the correct value. For example, as Figure 7 presents after the deprecated API `jaccard_similarity_score` got changed to `jaccard_score`, the default value of `average` is no longer valid. So RELANCER added that parameter in the call site with the correct value.

The size of the patches generated by RELANCER varied from 2 to 25 lines of code (LOC) with a median size of 3 LOC. When RELANCER was able to fix a notebook, the execution time varies from 2 seconds to 28 minutes, with on average two minutes (median 17 seconds) per notebook. This result demonstrates that RELANCER can be used in real time to fix deprecated errors.

Validation. We manually validated all the successful patches to make sure that RELANCER used the latest API and/or parameter for each deprecated API. Further, we also conducted an objective evaluation to make sure that the fixed notebooks serves the actual developer's intention. To this end, we leveraged the output cell feature of Jupyter Notebooks. More specifically, Jupyter Notebooks often have the output cell that stores the accuracy from the original run. We were able to identify the original accuracy from the output cell for 82 notebooks and found that the new accuracy we got after fixing the deprecation errors is within the 1% (median) of original accuracy. It should be noted that most machine learning models have inherent randomness. Therefore, we believe that such a small accuracy difference is not surprising.

```
-import plotly.plotly as py
+import chart_studio.plotly as py
py.iplot(data, filename=...)
```

Fig. 8: Example of moving an API to another library

Answer to RQ1: RELANCER restored execution to 56% of the notebooks, fixing as many as eight errors in a single notebook. The median time to fix a notebook is only 17 secs, making it suitable for real-time use.

B. RQ-2: Contribution of Different Sources of Information

To understand whether both sources of information: the API documentation and Github help RELANCER achieve its full capability, we created two baseline tools from RELANCER, which use only a single source of information along with error message. RELANCER$_{github}$ does not use any information from the API documentation and RELANCER$_{doc}$ does not use any information from GitHub. All other functionalities in two baselines are similar to RELANCER. Here we would like to stress the fact that since there is no tool available for upgrading APIs for Jupyer Notebooks, to the best of our knowledge, we cannot directly compare any existing technique. However, conceptually the baselines: RELANCER$_{github}$ and RELANCER$_{doc}$ roughly simulate the approaches that use only Github such as Meditor [27] and only the API documentation along with error messages such as SOAR [18].

The experimental results in Figure 6 show that RELANCER outperforms both RELANCER$_{github}$ and RELANCER$_{doc}$ significantly in terms of the number of fixed notebooks. More specifically, RELANCER fixed 142 notebooks while RELANCER$_{github}$ and RELANCER$_{doc}$ fixed 96 and 92 notebooks, respectively.

There may be two reasons why RELANCER$_{doc}$ cannot fix a particular bug. First, an API can be completely removed from a library. Figure 8 presents such an example where function `plotly.plotly` is completely removed from *plotly version 4* distribution package. According to the official documentation, it has moved to *chart-studio* [69]. Therefore, the replacement is not simply available in the latest version of the documentation of *plotly*. Second, many APIs move to different modules or the name changes so significantly that it is hard to create a mapping between the old API and the new API by a particular heuristic. For example, in our motivating example in Figure 1, the module `sklearn.grid_search` is deprecated and replaced by `sklearn.model_selection` starting from scikit-learn 0.18 [52]. For all these cases, Github can help rank the correct program element toward the top.

Similarly, not all possible example changes for deprecated APIs with their parameters are available on Github. For example, `tensorflow.train.RMSPropOptimizer` is deprecated and used by a notebook in our dataset, but our on-demand search did not provide any useful candidate. This may often happen for less popular APIs.

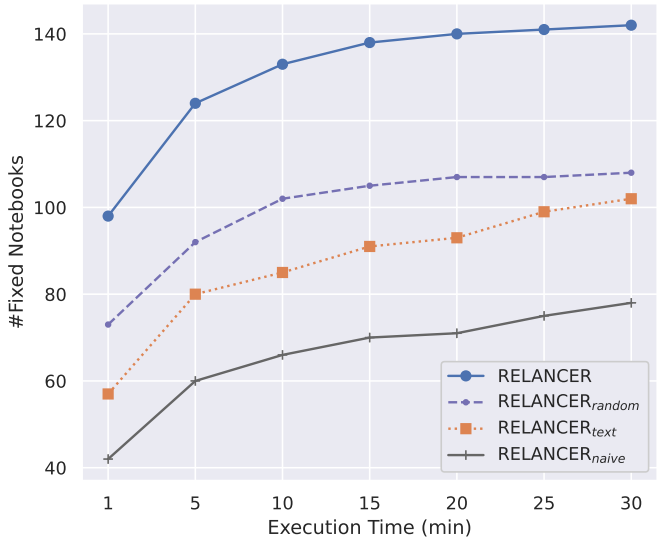

Fig. 9: Number of fixed notebooks over time.

In summary, all these results demonstrate that RELANCER cannot achieve its full potential without harnessing the information from multiple sources since each source plays an important role for a specific task.

Answer to RQ2: Both sources of information contribute to the overall effectiveness of RELANCER. RELANCER can successfully restore the executability of 56% of the subjects, while baselines relying on just GitHub examples and just API documentation can only fix 38% and 36% of the subjects respectively.

C. RQ-3: Contribution of Machine Learning Models

RELANCER uses two machine learned models to explore the search space of all the candidate program elements efficiently. However, it may be plausible to think that two simple static models can be used instead. Therefore, to better understand the contribution of these two models on the end-to-end performance of RELANCER: we have created three baselines:

RELANCER$_{random}$. In this baseline, RELANCER ranks the repair actions randomly instead of using its repair action model.

RELANCER$_{text}$. In this baseline, RELANCER's learning-to-rank model is replaced by an edit distance based ranking. More specifically, we combine the candidate elements from both GitHub and API documentation, and sort them by the Damerau–Levenshtein Distance score [66] between the deprecated element and the new element in descending order.

RELANCER$_{naive}$. In this baseline, we replace both machine learned models byRELANCER$_{random}$ and RELANCER$_{text}$ to understand the effect of two models in aggregation.

Figure 9 presents the experimental results in terms of the number of notebooks fixed by RELANCER and the baselines over their execution time. From the results, it is evident that even we replace one machine learned model with a simple

baseline model, the performance of RELANCER deteriorates. More specifically, RELANCER$_{random}$, RELANCER$_{text}$, and RELANCER$_{naive}$ fix 108, 102, and 78 notebooks, respectively, which is less than the 142 fixed by RELANCER. This indicates that both machine learned models contribute to the overall performance of RELANCER. We performed an unpaired t-test between the result of RELANCER with that of each baseline separately. The p-values of RELANCER vs RELANCER$_{random}$, RELANCER vs RELANCER$_{text}$, and RELANCER vs RELANCER$_{naive}$ are 0.0014, 0.0002, and $<$0.0001, respectively. The test result indicates that the RELANCER's performance improvement over any baseline is not just numerically but also statistically significant.

If we reduce the timeout of RELANCER and other baselines, as expected the number of notebooks fixed by each technique reduces. However, due to its machine learning models, the performance of RELANCER deteriorates proportionally less than the other baselines. If we set a timeout of 5 minutes for all the tools, RELANCER can fix 124 (49%) notebooks of notebooks while the naïve baseline can fix only 60 (24%) notebooks.

Answer to RQ3: The two machine learned models help RELANCER realize its real-time use case. RELANCER can fix 49% of subjects within 5 minutes, while a baseline lacking its machine learning models can only fix 24%.

D. Limitations and Threats to Validity

1) Limitations: RELANCER was unsuccessful in fixing 113 notebooks. To understand the reasons why RELANCER was unable to fix those notebooks, we manually investigated each of them and found one of the following reasons:

RELANCER is limited to fix only deprecation errors. We found that there are 55 notebooks that contain some other irrelevant errors such as missing dependent files, accessing incorrect columns in the dataset, and so on. Therefore, although RELANCER fixed the deprecation errors in these notebooks, finally the notebook still failed due to the other errors.

Lack of Type Inference. RELANCER determines the fully qualified name of an API based on a static analysis to upgrade it properly. However, since Python is a dynamically typed language, RELANCER cannot determine the type of some complex APIs, especially when it is dependent on the return type of another API. For example, the API: `as_matrix()` in Figure 10 has been deprecated [70]. To fix this API, RELANCER needs to know the fully qualified name: `pandas.DataFrame.as_matrix()` which should be determined by the returned type of `df.drop()`. Currently, RELANCER does not infer the return types of third-party APIs as the dynamic typing of Python makes it complex to retrieve accurate return-type information statically.

Lack of Complex Repair Actions. Currently, RELANCER's repair actions include replacing one API or variable name by another API or variable name respectively. However, some fixes require complex repair actions. For example, Figure 11 represents a deprecated API where the fix requires changing its

```
1 X = df.drop(['Radiation', ...], axis=1).as_matrix()
```

Fig. 10: A deprecation issue requires type inference to fix.

```
1 -scipy.misc.toimage(array)
2 +PIL.Image.fromarray(array.astype('uint8'))
```

Fig. 11: Example of a complex repair action.

parameter by another method call. Although RELANCER successfully changed the fully qualified name of the API, replacing the parameter `array` with the API `array.astype('uint8')` is currently out of its scope.

Change in Functionality. Figure 12 presents an example where due to a change in the functionality of the deprecated API: `tensorflow.placeholder`, the new API requires calling another API: `tf.compat.v1.disable_eager_execution()` with it. Currently RELANCER cannot infer such a change in functionality and thus is unable to fix this kind of errors.

2) External Validity: Our framework has only been instantiated for fixing API deprecation issues of Python-based Jupyter Notebooks, and has been only evaluated on 255 notebooks. Therefore, our results may not hold outside this scope. We tried to mitigate this risk by targeting real-world highly-voted datasets on Kaggle.

3) Quality of Data: The performance of RELANCER is limited by the quality of the training data. To mitigate this threat, we constructed the ground truth set from 12 most popular libraries and manually verified each deprecated API.

VI. RELATED WORK

Restoring the executability of Jupyter Notebooks. Recently, Wang et al. proposed Osiris [14], the first technique targeting unexecutable Jupyter Notebooks, specifically ones impacted by ambiguous or undefined order of cell execution. Osiris reconstructs possible execution orders by automatically satisfying dependencies between code cells. In follow up work they propose SnifferDog [15] that restores executability to a notebook by inferring and providing a compatible environment, comprised of correctly-versioned packages, for it. Our work complements [14] and [15] by remedying a third cause of notebook unexecutability, namely the deprecation of APIs. Moreover, in contrast to SnifferDog, which attempts to re-create the *original* execution environment of a notebook, RELANCER

```
1 -x=tf.placeholder(tf.float32,shape=[None,20],...)
2 +x=tf.compat.v1.disable_eager_execution()
3 +x=tf.compat.v1.placeholder(tf.float32,shape=[None,20],...)
```

Fig. 12: Example of functionality change.

ports it to the *user's* (and/or most recent) environment, in line with its target use-case of exploratory programming.

API Migration. Research on the broad topic of API migration goes back more than two decades [19], [20], [21], [22], [23], [24], [25], [26]. In early work by Chow et al. [19], a library maintainer is tasked with annotating API functions undergoing interface changes with migration rules. Catchup! [21] records API refactoring actions as a library maintainer evolves an API. The recorded migration rules [19] or API refactorings [21] are then used to migrate client applications. Such semi-automatic approaches entail additional effort by library maintainers, while RELANCER is completely automatic.

A second class of techniques automatically infer API mappings and/or migration refactorings. SemDiff [20] and LibSync [23] both mine API migration patterns across different versions of a library, such as renamed methods or changed parameters, from migrated client code. Zhong et al. [29] broaden the scope of API migration by inferring API mappings between related libraries in Java and C#. They do so by using textual similarity to align client code of the two libraries and then inferring corresponding APIs based on their common usage pattern in the aligned client code. Nguyen et al. [30], [31], [32] also infer API mappings between Java and C# libraries but use statistical machine translation instead. All the above techniques focus only on identifying API mappings. By contrast, RELANCER performs an end-to-end migration, including modifying the target code.

Recent work has proposed fully automated API migration techniques [27], [28], [18]. Meditor [27] targets automatic migration of a system from one library version to another. It mines migration-related code changes from open source repositories and instantiates them in the target system, in a context-sensitive manner to produce the migrated system, which is provided to the developer for review. Similarly, APPEVOLVE [28] automatically migrates Android apps by mining API changes from existing projects, applying them to the target app, and validating the patched app through differential testing. In emerging research, SOAR [18] proposes a technique to migrate data science programs from one library to another (e.g., TensorFlow [46] to PyTorch [71]). SOAR infers a likely API mapping from the documentation of the source and target libraries. Then it synthesizes candidates instances of the migrated program, using differential testing against the original (reference) program as an oracle, pruning the search space using logical constraints generated from the Python interpreter's error messages. Unlike the above, RELANCER organically integrates knowledge from two sources - migration examples and API documentation - to derive its search space. This feature is one of the keys to its performance (Section V-B). Further, APPEVOLVE and SOAR use differential testing against the reference implementation as a *strong* oracle to derive the correct migration. Meditor relies on human inspection for validation. In RELANCER's use case, there is no executable reference implementation. Thus, RELANCER uses a carefully orchestrated pair of machine learned models to effectively predict the correct migration.

Program Repair. Some elements of RELANCER's design are also inspired by the body of work on Automatic Program Repair (APR) [72], [73], [74], [75], [76], [77], [78], [79], [80]. APR techniques try to fix functional bugs in client code by implicitly or explicitly searching a space of program mutations for a patch, typically using a test suite as an oracle. At a high level RELANCER also searches a space of program modifications, but unlike APR techniques it needs to perform multiple API migrations (vs. fixing a single functional bug) and do so in real time, without the aid of a strong oracle like a test suite. Because of these key differences, RELANCER relies on machine learned models powered by multiple knowledge sources - runtime error messages, API migration examples on GitHub, and API documentation - as its primary vehicle for efficiently navigating the search space, rather than a test suite.

VII. CONCLUSION

Data scientists typically practice exploratory programming using computational notebooks, iteratively refining their code and actively trying to re-use solutions created by other data scientists. However, most publicly available notebooks cannot be executed. One of the prominent reasons is the deprecation of data science APIs. In this work, we proposed RELANCER, an automatic technique that restores the executability of broken Jupyter Notebooks, by upgrading deprecated APIs. RELANCER integrates an iterative runtime error driven approach with a combined search space sourced from API migration examples and API documentation. Both features are orchestrated through machine learned models. An evaluation of RELANCER on Kaggle notebooks showed that it is effective in restoring executability to 56% of the subjects, and that its error-driven approach, use of a combined search space, and its machine learned models all contribute to its efficacy.

VIII. ACKNOWLEDGMENTS

This work was partially supported by a gift from the Fujitsu Research of America, Inc. and National Science Foundation Grant No. CCF-1718903.

REFERENCES

[1] Wharton School of Business, University of Pennsylvania. (Accessed in 2021) What's driving the demand for data scientists? https://knowledge.wharton.upenn.edu/article/whats-driving-demand-data-scientist/.

[2] Forbes. (Accessed in 2021) Why data science is such a hot career right now. https://www.forbes.com/sites/quora/2017/10/25/why-data-science-is-such-a-hot-career-right-now/.

[3] Kaggle. (Accessed in 2021) Kaggle. https://www.kaggle.com.

[4] M. Beth Kery and B. A. Myers, "Exploring exploratory programming," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2017, pp. 25–29.

[5] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[6] Jupyter. (Accessed in 2021) Jupyter notebook. https://jupyter.org.

[7] J. M. Perkel, "Why jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.

[8] D. Toomey, *Jupyter for data science: Exploratory analysis, statistical modeling, machine learning, and data visualization with Jupyter*. Packt Publishing Ltd, 2017.

[9] J. Inc. (Accessed in 2021) We downloaded 10,000,000 jupyter notebooks from github – this is what we learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/.

[10] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *International Conference on Electronic Publishing*, 2016, pp. 87–90.

[11] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier, "The jupyter/ipython architecture: a unified view of computational research, from interactive exploration to communication and publication," in *AGU Fall Meeting Abstracts*, 2014, pp. H44D–07.

[12] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *International Working Conference on Mining Software Repositories*, 2019, pp. 507–517.

[13] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in *Conference on Human Factors in Computing Systems*, 2020, pp. 1–12.

[14] J. Wang, K. Tzu-Yang, L. Li, and A. Zeller, "Assessing and restoring reproducibility of jupyter notebooks," in *International Conference on Automated Software Engineering*, 2020, pp. 138–149.

[15] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of jupyter notebooks," in *International Conference on Software Engineering*, 2021, pp. 138–149.

[16] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *International Symposium on Foundations of Software Engineering*, 2020, pp. 233–244.

[17] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in *International Conference on Software Analysis, Evolution and Reengineering*, 2020, pp. 81–92.

[18] A. Ni, D. Ramos, A. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "Soar: a synthesis approach for data science api refactoring," in *International Conference on Software Engineering*, 2021, pp. 112–124.

[19] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *International Conference on Software Maintenance*, 1996, p. 359.

[20] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering Methodology*, vol. 20, no. 4, pp. 1–35, 2011.

[21] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *International Conference on Software Engineering*, 2005, pp. 274–283.

[22] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *International Conference on Software Engineering*, 2012, pp. 353–363.

[23] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[24] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *International Conference on Software Engineering*, 2008, pp. 471–480.

[25] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *International Conference on Software Engineering*, 2010, pp. 325–334.

[26] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[27] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in *International Conference on Program Comprehension*, 2019, pp. 335–346.

[28] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *International Symposium on Software Testing and Analysis*, 2019, pp. 204–215.

[29] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *International Conference on Software Engineering*, 2010, pp. 195–204.

[30] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *International Conference on Automated Software Engineering*, 2014, pp. 457–468.

[31] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *International Symposium on Foundations of Software Engineering*, 2013, pp. 651–654.

[32] ——, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *International Conference on Automated Software Engineering*, 2015, pp. 585–596.

[33] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in *International Database Engineering & Applications Symposium*, 2014, pp. 356–361.

[34] J. Cohen. (Accessed in 2021) Near real-time vs. real-time analytics. https://harperdb.io/blog/near-real-time-vs-real-time-analytics/.

[35] M. Kaggle. (Accessed in 2021) Meta kaggle. https://www.kaggle.com/kaggle/meta-kaggle.

[36] Creative Research Systems. (Accessed in 2021) The survey system. https://www.surveysystem.com/sscalc.htm.

[37] Jupyter. (Accessed in 2021) nbconvert. https://github.com/jupyter/nbconvert.

[38] Anaconda. (Accessed in 2021) Anaconda. https://anaconda.org.

[39] scikit learn. (Accessed in 2021) scikit-learn: machine learning in python. https://scikit-learn.org.

[40] pandas. (Accessed in 2021) pandas – python data analysis library. https://pandas.pydata.org.

[41] seaborn. (Accessed in 2021) seaborn: statistical data visualization. https://seaborn.pydata.org.

[42] NumPy. (Accessed in 2021) Numpy. https://numpy.org.

[43] SciPy. (Accessed in 2021) Scipy. https://www.scipy.org.

[44] XGBoost. (Accessed in 2021) Xgboost: extreme gradient boosting. https://github.com/dmlc/xgboost.

[45] Plotly. (Accessed in 2021) Plotly: The front end for ml and data science models. https://plotly.com.

[46] TensorFlow. (Accessed in 2021) Tensorflow. https://www.tensorflow.org.

[47] Keras. (Accessed in 2021) Keras: the python deep learning api. https://keras.io.

[48] statsmodels. (Accessed in 2021) statsmodels. https://www.statsmodels.org.

[49] imbalanced learn. (Accessed in 2021) imbalanced-learn. https://github.com/scikit-learn-contrib/imbalanced-learn.

[50] CatBoost. (Accessed in 2021) Catboost - open-source gradient boosting library. https://catboost.ai.

[51] Kaggle. (Access in 2021) Predicting boston house prices. https://www.kaggle.com/sagarnildass/predicting-boston-house-prices.

[52] scikit learn. (Accessed in 2021) Scikit documentation. https://scikit-learn.org/stable/whats_new/v0.18.html.

[53] ——. (Accessed in 2021) sklearn.svm.linearsvc. https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html.

[54] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *European Conference on Machine Learning*, 1998, pp. 137–142.

[55] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

[56] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. M. Lopez, "The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in *Conference on Computer Vision and Pattern Recognition*, 2016, pp. 3234–3243.

[57] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig, "Virtual worlds as proxy for multi-object tracking analysis," in *Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4340–4349.

[58] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *European Conference on Computer Vision*, 2016, pp. 102–118.

[59] BeautifulSoup. (Accessed in 2021) Beautifulsoup. https://www.crummy.com/software/BeautifulSoup.

[60] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[61] GitHub. (Accessed in 2021) Github rest api. https://docs.github.com/en/rest.

[62] ——. (Accessed in 2021) Search. https://docs.github.com/en/rest/reference/search.

[63] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, "Autopandas: neural-backed generators for program synthesis," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2019, pp. 1–27.

[64] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 234–246, 2014.
[65] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.
[66] Wikipedia. (Accessed in 2021) Damerau–levenshtein distance. https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance.
[67] LightGBM. (Accessed in 2021) Lightgbm. https://lightgbm.readthedocs.io/en/latest/.
[68] LibCST. (Accessed in 2021) Libcst. https://github.com/Instagram/LibCST.
[69] plotly. (Accessed in 2021) Version 4 migration guide in python. https://plotly.com/python/v4-migration.
[70] pandas. (Accessed in 2021) What's new in 1.0.0. https://pandas.pydata.org/docs/whatsnew/v1.0.0.html#deprecations.
[71] PyTorch. (Accessed in 2021) Pytorch. https://pytorch.org.
[72] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*, 2009, pp. 364–374.
[73] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *International Conference on Software Engineering*, 2013, pp. 802–811.
[74] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multi-line program patch synthesis via symbolic analysis," in *International Conference on Software Engineering*, 2016, pp. 691–701.
[75] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *International Conference on Automated Software Engineering*, 2017, pp. 648–659.
[76] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *International Conference on Software Engineering*, 2019, pp. 13–24.
[77] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *International Symposium on Foundations of Software Engineering*, 2019, pp. 613–624.
[78] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.
[79] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.
[80] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

How Data Scientists Use Computational Notebooks for Real-Time Collaboration

APRIL YI WANG*, The University of Michigan, USA
ANANT MITTAL, The University of Michigan, USA
CHRISTOPHER BROOKS, The University of Michigan, USA
STEVE ONEY, The University of Michigan, USA

Effective collaboration in data science can leverage domain expertise from each team member and thus improve the quality and efficiency of the work. Computational notebooks give data scientists a convenient interactive solution for sharing and keeping track of the data exploration process through a combination of code, narrative text, visualizations, and other rich media. In this paper, we report how synchronous editing in computational notebooks changes the way data scientists work together compared to working on individual notebooks. We first conducted a formative survey with 195 data scientists to understand their past experience with collaboration in the context of data science. Next, we carried out an observational study of 24 data scientists working in pairs remotely to solve a typical data science predictive modeling problem, working on either notebooks supported by synchronous groupware or individual notebooks in a collaborative setting. The study showed that working on the synchronous notebooks improves collaboration by creating a shared context, encouraging more exploration, and reducing communication costs. However, the current synchronous editing features may lead to unbalanced participation and activity interference without strategic coordination. The synchronous notebooks may also amplify the tension between quick exploration and clear explanations. Building on these findings, we propose several design implications aimed at better supporting collaborative editing in computational notebooks, and thus improving efficiency in teamwork among data scientists.

CCS Concepts: • **Human-centered computing → Empirical studies in collaborative and social computing**; *Empirical studies in HCI*.

Additional Key Words and Phrases: computational notebooks, collaborative systems, data science

ACM Reference Format:
April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 39 (November 2019), 30 pages. https://doi.org/10.1145/3359141

1 INTRODUCTION

The complexity of data science work and the demand to adopt data science practices in various domains has grown rapidly in the last decade. With this increase in adoption, there is a need to facilitate collaboration among data science workers, domain experts, and consumers. Data scientists

*This is the corresponding author

Authors' addresses: April Yi Wang, The University of Michigan, 105 S State St, Ann Arbor, MI, 48103, USA, aprilww@umich.edu; Anant Mittal, The University of Michigan, 105 S State St. Ann Arbor, MI, 48103, USA, anmittal@umich.edu; Christopher Brooks, The University of Michigan, 105 S State St. Ann Arbor, MI, 48103, USA, brooksch@umich.edu; Steve Oney, The University of Michigan, 105 S State St. Ann Arbor, MI, 48103, USA, soney@umich.edu.

2573-0142/2019/11-ART39 $15.00
https://doi.org/10.1145/3359141

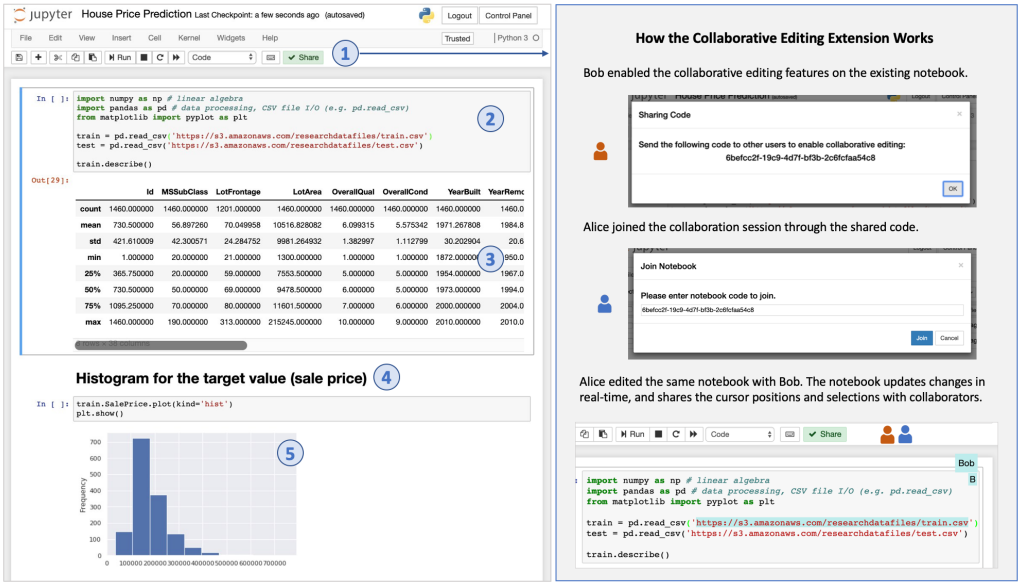

Fig. 1. An example of a Jupyter notebook. (1) A custom collaborative extension for users to share their notebook. (2) A notebook cell that contains code to import libraries and load dataset. (3) An output of a shared data frame. (4) A markdown cell that contains narrative text. (5) An output of a visualization

often create *computational narratives*, which combine data, code to process those data, and natural language explanations to form a narrative. Some even consider computational narratives to be the engine of collaborative data science [32]. *Computational notebooks* allow data scientists to create and share computational narratives. *Jupyter Notebook*[1], a computational notebook platform that supports more than 40 programming languages, has been widely used for writing and sharing computational narratives in various contexts [59]. For example, data science instructors use Jupyter notebooks to create interactive lecture notes or textbooks. Data science learners can experiment with these interactive lecture notes to deepen their understanding or explore alternative solutions [45]. Researchers use Jupyter notebooks to demonstrate their computational work and share their data analysis process for open science, which makes it easy for others to reproduce the results [61]. As Figure 1 shows, Jupyter notebooks allow users to weave together source code, narrative text, visualization, computational outputs, and other rich media using structured cells.

Prior studies have revealed the challenges in constructing and sharing computational narratives through notebooks. For example, data scientists are reluctant to keep up-to-date explanatory notes, which impedes sharing and collaboration [63]. Studies have also explored ways to lower the barriers for writing and sharing computational narratives: folding content selectively [62]; local version control mechanisms [36]; and managing and reorganizing content [28]. Most of these innovations are designed and evaluated for sharing the computational narrative after it is finished, leaving data scientists to work on individual notebooks. Recently, tools like Google Colab[2] have demonstrated the possibility for *synchronous editing* — multiple users are able to edit the same notebook and changes are updated in real-time, which may revolutionize the ways data scientists collaborate.

[1] https://jupyter.org
[2] https://colab.research.google.com

However, synchronous editing comes with its own challenges and may not always improve work efficiency. Studies have identified several issues with synchronous editing in other contexts. For example, in narrative writing, multiple users rarely synchronously edit adjacent content [11], and programmers can interfere with each others' work when using synchronous code editors [21]. While many of these studies are constructed around the context of collaborative writing and programming, it remains unknown how synchronous editing in computational notebooks might work for data scientists. More specifically, we are interested in three questions:

(1) What tools and strategies do data scientists currently use for collaboration?
(2) Compared to working on individual notebooks in a collaborative setting, how does synchronous notebook editing change the way data scientists collaborate in computational notebooks?
(3) What challenges, if any, do data scientists perceive in synchronous notebook editing?

To address these questions, we first conducted a formative survey with 195 data scientists who were familiar with Jupyter notebooks. Based on the common tools and mechanisms they use for collaboration and communication, we further conducted an observational study with 24 intermediate data scientists working in pairs remotely to solve a predictive modeling problem. To understand a broader spectrum of synchronous notebook editing, we assigned participants with different settings for collaboration, which included: (1) working on *synchronized notebooks* — participants worked on notebooks with synchronous editing similar to that in Google Docs, and could communicate through other tools, (2) working on *individual notebooks* — participants were not able to edit the same notebook, but could communicate the intermediate code and output through other tools. We also provided participants different options for communication which included: (1) text-based messaging, (2) video chat. Through this observational study we empirically probed both the benefits and challenges for the different ways of collaborating with computational notebooks.

Our key finding reveals that working on synchronized notebooks can improve the collaboration outcomes by reducing communication costs and encouraging more exploration in a shared context. However, working on synced notebooks requires participants to be more strategic when coordinating their work. Working on synced notebooks is more likely to lead to unbalanced participation where one team member does the majority of implementations and ideation. In addition, participants found other challenges in using synchronized notebooks such as interference with each other, lack of awareness, and privacy concerns. These findings suggest ways in which we need to improve the design of collaborative notebook editing tools to better foster teamwork among data scientists.

The main contributions of this work are the empirical insights we present on how data scientists collaborate using computational notebooks. These insights lead to design implications to enhance collaborative computational notebooks for fostering collaboration among data science learners and practitioners. This work extends prior work on real-time collaborative systems and is broadly applicable to understanding collaboration in exploratory and open-ended tasks.

2 RELATED WORK

To contextualize our studies, we draw upon prior research in collaborative data science work practices, tools and systems that extend the usability of computational notebooks, computer-mediated collaboration, and its practice in the context of writing and programming.

2.1 Data Science Applications and Processes

The term *data science* was first distinguished from pure statistics in 1997, with the notion of extracting knowledge and insights from data [14]. The field of data science has grown rapidly over the last decade amidst the rise of big data and breakthroughs in technologies like machine learning that expand our capabilities for understanding data [12]. According to a survey of the United States workforce on LinkedIn [1], the demand for data scientists will continue to increase as more industries (e.g., finance, business, healthcare) adopt big data to make business decisions.

The process of doing data science has been categorized and discussed among statisticians, computer scientists, HCI researchers, and others (e.g., [22, 34, 40, 52, 55]). O'Neil and Schutt distinguished the data science process into several iterative phases [55]:

(1) *Collecting* data from a variety of sources (e.g., emails, logs, and medical records)
(2) Building and using pipelines for *data munging* (e.g., joining, scraping, and wrangling)
(3) *Cleaning* data to ensure its validity and accuracy for analysis (e.g., manipulating duplicates, filtering outliers, and tuning missing values)
(4) *Exploring and hypothesizing* the relationship between variables using different techniques (e.g., generating statistical summary, plotting pairwise relationships)
(5) *Applying machine learning algorithms* or statistical models based on the type of problems (e.g., k-nearest neighbor, linear regression, and naive Bayes)
(6) Finally *interpreting and communicating* results to different audiences (e.g., managers, co-workers, and clients)

Not all stages are required depending on the type of data science task. The exploration process can be non-sequential, as feedback from later stages may result in additional work in earlier stages. Kery and Myers [37] considered data science as a practice of *exploratory programming* where programmers write code to experiment with ideas in an open-ended task. In particular, communication can happen back and forth in a collaborative setting, while collaboration patterns may vary between stages.

2.2 Collaboration in Data Science

Prior research has found that data scientists in software companies often work collaboratively [40]. For example, some data science teams have adopted a triangular structure where they divide the task into collecting data, cleaning data, and analyzing data. However, collaboration in data science can be challenging. Transferring findings from data science work to business actions requires successful communication between data scientists and stakeholders who are usually non-technical professionals [57]. Kandel et al. [34] revealed that collaboration between data scientists rarely happens in domains like marketing and finance. One major reason is that the diversity of tools and programming languages has made it laborious to share intermediate code, especially when it is not well documented. Correspondingly, Kery and Myers [37] addressed the difficulty of maintaining a shared understanding of the exploration progress since the intermediate code and data artifacts can be experimental and messy. Nonetheless, with the growing demand for data analysis, efficient collaboration between data scientists will become increasingly important and challenging. Building upon the work of Kandel et al. [34], and Kery and Myers [37], our study further investigates the benefits and trade-offs of mechanisms that enable real-time collaboration and communication in data science work.

2.3 Computational Narratives, Computational Notebooks, and Innovations

Data science involves a large amount of experimentation and subjective decision making. Thus, it is important for data scientists to document the story behind the computation of results (e.g.,

reporting alternative solutions and explaining the limitations for them). A variety of media such as textual explanations, graphs, forms, interactive visualizations, code segments, and output are used for narrative and storytelling to document the data analysis process. Extending Knuth's idea of literate programming, wherein he argues that making programming sources more understandable to human beings can be achieved by combining a programming language with a documentation language [41], computational narratives serve as literature for data scientists and data science consumers to present work and exchange ideas.

2.3.1 Computational Notebooks. Researchers and practitioners have long explored approaches for creating and sharing digital documents for data analysis (e.g., [9, 24, 66]). Establishing a common format for documenting data analysis can make it easier to present, reproduce, share, and collaborate. One approach is to capture digital assets (e.g., code, output, documentation) and computational environments (e.g., browsing, UI interaction, file versioning) from the OS level [24]. Another approach integrates digital components (e.g., text-based lab notes, emails, web pages) into a combined entry [66]. Yet, the most popular tools for data scientists are computational notebooks — web-based platforms that allow users to write and execute code, inspect output, and integrate text annotations, figures, interactive visualization, and other rich media (e.g., Apache Zeppelin[3], Spark Notebook[4], Observable[5], and Jupyter Notebook).

2.3.2 Jupyter Notebook. Project Jupyter evolved from IPython [58], a terminal-based interactive shell that originally designed for creating interactive visualizations for scientific computing. Wrapping IPython as the kernel, Project Jupyter is designed as a web-based platform for authoring a single document that combines code cells and intermediate results. The evolution of Project Jupyter is influenced by the rise of data science. Data science is exploratory and fluid, and the process benefits from creating reproducible computational narratives for iterative exploration and interactive inspect of intermediate results. Jupyter Notebook is an open source project that is also extensible through optional add-ons. As Figure 1 shows, Jupyter notebooks consist of "cells" — typically small chunks of code or narrative text in the Markdown format. Users can execute cells (typically, but not necessarily, from top to bottom) and observe their outputs, which can include visualizations, data frames, or rendered narrative text.

Most other notebook platforms have a very similar user interface to Jupyter but differ in the programming languages they support (Jupyter uses Python by default but its architecture can support other languages). Other computational notebook platforms include Observable (which uses JavaScript), RStudio[6] (which uses R Markdown), Wolfram Notebooks[7] (which uses the Wolfram Programming Languages), and Zeppelin (which allows multiple programming languages to be used in the same notebook). Some notebook platforms use a different computational architectures. For example, some notebook platforms enable *reactive* notebooks that automatically run cells when necessary (as opposed to requiring that users run cells manually, as Jupyter does). For example, Observable notebooks run cells in "topological" order—data dependencies between cells are tracked and changing a cell automatically re-runs other cells that depend on its result. Compared to other notebook services, Jupyter has a larger community of users given its' long history and prevalence among different contexts. Jupyter also has more customized extensions because of its large community.

[3] https://zeppelin.apache.org
[4] http://spark-notebook.io
[5] https://observablehq.com
[6] https://www.rstudio.com/
[7] http://www.wolfram.com/notebooks/

Jupyter has become one of the most popular tools for data related work among academia, industry, and data science education [59], even earning the 2017 ACM Software System Award [33]. Studies have found that Jupyter is popular in a variety of settings. Kross and Guo [45] interviewed practitioners who taught data science and found that Jupyter notebooks have been widely used by instructors to deliver course materials. They also found that Jupyter notebooks allow students to easily write computational narratives with a low cost for setting up an environment. Kery et al. studied how professional data scientists used Jupyter notebooks in their daily work to create computational narratives [39]. Randles et al. [61] investigated how Jupyter notebooks can be used for open science under the principles of Findable, Accessible, Interoperable, Reusable (FAIR). In fact, some academic venues encourage paper authors to include notebooks with their submissions (e.g., the Distill Journal[8] in the area of machine learning).

However, studies have identified several limitations with computational notebooks [39, 63, 64]. Rule et al. conducted a large scale analysis of over 1 million open-source computational notebooks and found that only one in four held explanatory text [63]. Kery et al. interviewed 21 data scientists to study their coding behaviors using computational notebooks and highlighted the challenges of tracking history of experimentation [39]. Both studies revealed the tension between using computational notebooks for rapid exploration and instructive explanation. For quick exploration, data scientists sometimes generate messy and informal notebooks, which can be difficult for others (or even the author) to read later on [63]. Data scientists have to use strategies like actively pausing the experiment to curate and clean notebooks into narratives, which may hinder the exploration process [39]. The tension between quick exploration and instructive explanation can be contextually sensitive depending on how exploratory and open-ended the task is. Building upon current computational notebooks, other systems have explored designs to better support non-linear exploration in notebooks. For instance, Kery et al. integrated a lightweight local versioning mechanism to help data scientists keep track of their exploration history [35, 36, 38], while Rule et al. took a different approach to enable data scientists to fold content blocks with annotations in notebooks [62]. Head et al. explored code gathering extensions for data scientists to manage and navigate through cluttered and inconsistent notebooks [28], and Zhang and Guo [73] created *DS.js* to transform any webpage to a computational notebook, lowering the barrier for novices to retrieve data for exploration. Companies have also built innovations on the notebook infrastructure to support various data related tasks in practice. For example, the Netflix data team [5] developed nteract[9], with extra features for data explorations; Papermill[10], which facilitates rapid exploration by spawning notebooks for different parameter sets; and Commuter[11] — a platform for curating and sharing notebooks.

The tools mentioned above focus on the use of computational notebooks for individual authoring. Recently, tools like Google Colab[12] have further fostered collaborative data science by allowing multiple users to edit the same notebook in real time. However, prior studies on computational notebooks have only inspected individuals authoring the notebooks, not yet multiple data scientists collaboratively authoring the notebooks. Only recently, Koesten et al. [42] interviewed data practitioners about their collaborative practices with structured data. They synthesized collaboration needs across a wide range of scenarios from co-creation data analysis to reusing others' data in a new context. Our work takes a different scope to investigate real-time collaborative editing in

[8] https://distill.pub/journal
[9] https://github.com/nteract/nteract
[10] https://github.com/nteract/papermill
[11] https://github.com/nteract/commuter
[12] https://colab.research.google.com

computational notebooks. We aim to understand whether collaborative editing would help collaborators maintain a shared understanding or rather intensify the tension between exploration and explanation. In our discussion, we propose suggestions to improve the current design of real-time collaborative editing features to better support collaborative data science.

2.4 Collaborative Systems

Since the late 1980s, work has explored different forms of computer-mediated collaboration [44, 48, 51], evaluations of collaborative systems [27, 46], and principles for designing computer-based collaborative tools [18, 43]. One critical aspect for collaborative systems is supporting group awareness [15, 25, 46]. Presenting awareness information in a group facilitates collaboration by maintaining a shared mental model among collaborators, by reducing communication costs, and thus improving the usability of real-time distributed groupware systems [25]. To aid the design of groupware systems, Gutwin and Greenburg have specifically examined collaboration in small groups and proposed a conceptual framework to describe various aspects of workplace awareness [26]. The framework has addressed how people gather and use workplace awareness information for collaboration and issues around awareness information. Our studies also shed light on the common awareness mechanisms (e.g., shared cursor information, shared editors) in real-time collaborative notebooks. Existing real-time collaborative notebooks such as Google Colab migrate these common awareness mechanisms from collaborative writing systems like Google Docs. However, these mechanisms may be limited for collaborative data science given the differences between the nature and complexity of tasks. For example, data scientists may want more awareness information on whether their collaborators are modifying a *data frame* — the instance of data in a structured format. To understand how these mechanisms support collaboration in other contexts, we surveyed studies on collaborative writing, coding, and visual analytics practice below.

2.4.1 Collaborative Writing. With the emergence of early collaborative writing systems (e.g., SASSE [2, 50], ShrEdit [53]), researchers have developed theoretical frameworks and taxonomies for issues around collaborative writing [49, 60]. Posner and Baecker synthesized common roles, activities, and writing strategies in collaborative writing [60]. In particular, they have discussed five writing strategies in the joint writing process:

- Single writer: one team member writes the document with minimal assistance from others
- Scribe strategy: one team member writes the document while the others engage in the ideation process
- Separate writers: dividing the document into parts and have different individuals writing the various parts
- Joint writing: group members write the document closely together
- Consulted strategy: a combination of the other writing strategies that involves with a consultant throughout the project

More recent studies have revisited how people write together using modern collaborative writing tools like Google Docs (e.g., [4, 11, 54, 69, 72]). For example, Yim et al. [72] explored styles of synchronous writing by analyzing 45 Google Docs documents and found that students tended to produce higher quality documents when they worked in divide and conquer mode with balanced participation.

A recent analysis of usage patterns in Etherpad (a collaborative editing system) revealed that users rarely edited near-by document parts simultaneously in practice [11]. For instance, roughly half of the Etherpad documents retrieved from a public platform only had one author. For the other half, in most cases, authors took turns editing documents asynchronously. Correspondingly, Wang et al. interviewed 30 users and surprisingly found that users were reluctant to write together when they

were collaboratively constructing a document [69]. For example, role structure and concomitant power differentials may hold users back from writing together; the co-editing environment may blur the accountability and credit of contribution; there might be a social embarrassment to be watched by others when typing. These findings triggered our interest in understanding how data scientists perceived writing a computational narrative in a collaborative setting. Furthermore, studies on collaborative editing inspired our study design to observe collaboration in a given time limit (e.g., [72], [4]). Although data scientists may not always need to work in the same notebook simultaneously in the real world, understanding and supporting synchronous collaboration on computational notebooks is still valuable and important as a first step to understand other forms of collaborations.

2.4.2 Collaborative Programming. Collaborative programming among geographically distributed teams is a broadly studied topic of Software Engineering and HCI research. Built upon discussions on coordination and management strategies in software engineering process (e.g., [3, 30, 65, 68]), researchers have explored various technical interventions that enable real-time collaborative programming (e.g., [7, 8, 13, 16, 21, 23, 47, 56, 70]). For example, the system ATCoPE integrated version control support in real-time collaborative Integrated Development Environments (IDEs) [16]; Lee et al. explored the subtasks model for collaborative code editors [47]. Studies also addressed issues that impeded collaboration. Apart from what is discussed in collaborative writing practice [11, 69], Goldman addressed a problem that one person's compilation error may completely block others' work [20]. Goldman further examined different roles in collaborative programming [19], and designed an asymmetric, role-based environment to alleviate conflicts in real-time collaborative software development [21].

Wilson et al. [71] studied the benefits of pairing for programming tasks and found it led to better problem-solving. *Pair programming*—where two programmers work together at one work station and take turns to write and review code—later became popular in software development practice [6]. Another common scenario for real-time collaborative programming is tutoring, where tutors help learners with debugging through shared computational environments. Warner and Guo integrated features like bug reporting and version control management in the real-time collaborative system CodePilot [70] to help novice programmers. Guo designed Codeopticon [23], which allowed tutors to monitor the programming environments of multiple learners at scale. Codeon [7, 8] embedded an on-demand help requests component within an IDE and shared related code contexts to remote helpers. Oney et al. further introduced a deictic code referencing module that tracks code history and allows users to make inline references for chat messages [56]. These systems have all been designed or evaluated in the educational context. Similarly, our study of supporting real-time collaborative editing in computational notebooks can explicitly benefit data science novices and students.

2.4.3 Collaborative Visualization. Visual analytics, which uses visual information to support sense-making, is a powerful tool for data science. Most visual analysis systems (e.g., Tableau[13]) are designed to support code-free exploration through interactive user interfaces. Studies in the area of information visualization have shed light on the benefits and challenges of collaborative visualization. As defined by Isenberg et al. [31], collaborative visualization is "the shared use of visual representations of data by multiple users with the goal of joint information processing". Collaborative visualization can be distinguished from collaborative data science as it highlights the perspective on joint viewing, interacting, discussing and interpreting the data representation.

[13] https://www.tableau.com

Visualization research has also explored different types of collaboration scenarios: collocated collaboration through large displays and shared workspaces, distributed collaboration through real-time shared displays. In particular, Heer and Agrawala [29] discussed issues for designing asynchronous collaboration systems for visual analytics (e.g., building common ground and awareness, designing reference and deixis, maintaining group dynamics, and establishing group consensus).

3 OVERVIEW OF THE METHODOLOGY

This project investigates three research questions:

RQ1 What tools and strategies do data scientists currently use for collaboration?
RQ2 Compared to working on individual notebooks in a collaborative setting, how does synchronous notebook editing change the way data scientists collaborate in computational notebooks?
RQ3 What challenges, if any, do data scientists perceive in synchronous notebook editing?

To understand tools and strategies data scientists currently used in practice (RQ1), we first conducted a survey with 195 data scientists/data science students who came from diverse backgrounds. We summarized the tools they used for programming, communication, and project management, as well as their strategies for collaboration. In particular, we identified two approaches for data scientists to collaborate: 1) the traditional collaboration setting where team members work on individual Jupyter Notebook and update each others' work asynchronously, and 2) the emerging collaboration setting where team members work closely together on a shared Jupyter Notebook and all the edits are synchronized in real-time.

To further compare how data scientists' collaboration styles varied between two approaches (RQ2), we conducted an observational study with 24 intermediate data scientists working in pairs remotely to solve a predictive modeling problem. We summarized the common collaboration styles that emerged in the two collaboration settings. We reported the comparison of communication styles, performance, and perceptions of the collaboration experience. We also analyzed the challenges that participants faced in using real-time collaborative editing features (RQ3).

4 STUDY 1: FORMATIVE SURVEY ON COLLABORATIVE DATA SCIENCE

We conducted a formative survey to investigate data scientists' previous experiences with collaboration. In particular, we aimed to understand the tools they used for programming, communication, and project management, as well as their strategies for collaboration.

We sent the survey to data science interest groups in a university and to individuals who completed a specialization course on Coursera that teaches data science using Python. Respondents had to meet the following criteria to take the survey: (1) be familiar with Python and Jupyter notebooks, (2) have formal training in data science, and (3) have worked individually on at least one data science project. To motivate participation, we randomly picked two respondents and rewarded them each a $25 gift card.

4.1 The Survey Instrument

The online survey consisted of four sections: (1) informed consent, (2) demographics and data science experience, (3) experience with collaborative data science, and (4) willingness to participate in the observational study.

In the first section, we explained the purpose of the survey and collected respondents' consent. The second section asked about demographics (age, gender, educational background, job role), experience with data science (e.g., what techniques they have used or learned in the past), and degrees of identification as the given roles: computer scientists, software engineers, data scientists,

and statistician analysts. In the third section, we asked respondents to recall their most recent experience of working with other people on a data science project. If respondents did not have any experience in collaboration, we provided them a hypothetical scenario:

> "Imagine you are participating remotely at a 2-day long data science hackathon with 2 other team members on a predictive modeling problem."

Respondents were then asked to provide more details about the project, for example, describing the context of the project (e.g., purpose, problem, and dataset), and selecting activities that are involved in the project (e.g., collecting data, sampling data, feature selection, data visualization). In addition, respondents were asked to list the tools they have used when collaborating with others on data science projects for the purposes of *programming*, *communication*, and *project management*. Followed by an open-ended question, we asked their strategies for keeping a shared understanding across the team. In the last section, we asked whether respondents would be willing to participate in an observational study. We collected the names and e-mail addresses of participants that indicated they were willing to participate.

4.2 Data Analysis

We first filtered the responses by deleting incomplete entries and merging duplicated entries from the same respondents. For the open-ended questions, we took an inductive analysis approach to explore themes for each question. Four coders worked individually on a small sample of the responses and developed a list of potential codes. After discussing and merging the coding scheme, we coded the same sample independently. We then compared the result by computing a Fleiss' Kappa score to measure both the reliability of the coding scheme and the agreement among the coders. We iterated on the coding scheme until an appropriate level of agreement was achieved among the four coders (Fleiss's kappa, $\kappa = 0.74$).

5 KEY FINDINGS FROM STUDY 1

We reported the key findings from the formative survey study.

5.1 Data Overview

The survey received 195 valid responses in total (23.08% female and 76.92% male). Among 195 responses, 35 are from data science interest groups in a university and 160 are from individuals who completed a data science specialization course on Coursera. The majority of respondents (73.8%) were age 20–40. Respondents came from a variety of job roles: students (29.74%), data scientists (25.13%), software engineers including Information Technology (IT), system architecture (14.87%), researchers (9.74%), managers including CEOs, VPs, and product managers (9.23%), business analysts (8.20%) and others (3.09%, e.g., drilling engineer).

The respondents were generally well trained in data science. The majority of them (94.36%) held or were pursuing a bachelor or higher degree. Most of these degrees are in technical fields (e.g., computer science, information science, electrical engineering, applied science, or data science). When asked about their previous experience in applying common techniques in data sciences, the respondents indicated they were skilled in linear regression (96.92%), decision trees (92.31%), SVMs (85.64%), neural networks (82.05%), non-linear regression (70.26%), deep learning (65.13%), Bayesian modeling (62.56%), and other advanced techniques (e.g., XGBoost, reinforcement learning).

The respondents also reported engaging in a variety of data science activities: data cleaning (67.69%), applying machine learning algorithms (57.95%), data visualization (52.82%), exploratory data analysis (49.74%), collecting raw data (44.10%), writing a report (38.97%), feature selection

Purpose	Tool	Percent
Programming	Jupyter Notebooks	88.72%
	IDEs (e.g., RStudio, PyCharm)	51.79%
	Code Editors (e.g., Atom, Sublime)	46.15%
	Google Colab	12.31%
Communication	E-mail	79.49%
	Face-to-face Communication	68.72%
	Instant Messaging	55.90%
	Google Docs	40.51%
	Video Conferencing	38.46%
Project Management	Version Control (e.g., Github, Bitbucket)	49.74%
	Task Tracking (e.g., Trello, Jira)	21.03%
	Shared Storage Services (e.g., Google Drive, Dropbox)	3.07%

Table 1. The tools that respondents have used for programming, communication and project management during collaboration.

(38.46%), applying statistical models (31.28%), doing an oral presentation (31.28%), data sampling (28.20%), hypothesis testing (22.56%), and building data products (21.03%).

5.2 Experience with Collaborative Data Science

When asked about their previous experience in collaborative data science, most respondents (73.3%) reported that they had prior experience collaborating with other people on a data science project. Of the respondents who had previous collaboration experience, most (72.72%) collaborated on a data science project for work, while the others mentioned other purposes such as course projects (30.07%), competitions or hackathons (18.89%), or personal side projects (13.20%).

5.2.1 Choices of Tools. We asked respondents to list the tools they have used for programming, communication and project management during collaboration. For respondents who do not have or can not remember their past collaboration experience (26.7%), we gave them a scenario of participating in a data hackathon in teams and asked about their choices of tools and collaboration strategies. For programming purposes, most respondents mentioned Jupyter notebooks (88.72%), IDEs (51.79%), and code editors (46.15%). In addition, some respondents mentioned Google Colab (12.31%), a computational narrative environment which is an alternative to, but similar to, Jupyter notebooks. For communication purposes, e-mails (79.49%) and face-to-face communication (68.72%) are most mentioned by respondents, followed by instant messaging (55.90%), Google Docs (40.51%) and video conferencing tools (38.46%). For project management tasks, roughly half of respondents mentioned version control suites (49.74%) such as Github and Bitbucket for sharing code and datasets. Several respondents also mentioned using shared storage services such as Google Drive and Dropbox for managing project assets. Some respondents mentioned task tracking tools (21.03%) such as Trello and Jira. Respondents also mentioned using Google Calendar, spreadsheets, or physical whiteboards to keep track of tasks and project progress. Our results are summarized in Table 1.

5.2.2 Strategies for Keeping a Shared Understanding. When asked about their strategies for keeping a shared understanding across team members, most respondents mentioned regular discussions

Strategy	Percent	Example Response
Discussions and meetings	54.36%	*There were weekly meeting among team members to keep track of the progress of each element of the project*
Frequently check-ups	51.79%	*Communicate actively and frequently; check-up on every hour*
Documentation	48.20%	*Keep notes in Google Docs; ... comments in code;*
Organization	28.72%	*Divide up the work into definable parts, make sure that everyone knows the progress that everyone else has made and how it impacts their part of the project*
Shared Assets	25.13%	*Common repository for files; share the same place for storing project, and name the file clearly*
Others	5.01%	*Code review to ensure code matched intent; all worked in a friendly manner*

Table 2. Strategies for keeping a shared understanding

and project meetings (54.36%). For example, respondents mentioned "weekly meetings with the team to follow up the stages of deployment", and also frequently reported that they used check-ups (51.79%) such as keeping the other team members informed of any changes made to the code. Some of these respondents reported that they would work closely in a physical space to reduce the communication cost by talking face-to-face. Documenting (48.20%) is another common strategy used for collaboration. For example, respondents mentioned that they would keep all the intermediate findings in shared Google Docs. Some respondents also mentioned coordination strategies such as planning ahead and being clear about everyone's responsibility (28.72%). In addition, respondents mentioned that they would share any intermediate results and code using version control tools or shared folders (25.13%). The others mentioned strategies such as code reviews to help them maintain a shared understanding. The results are summarized in Table 2.

5.3 High-Level Summary of Findings

In summary, respondents to our survey were made up of a variety of practitioners and students who are well trained in data science and are familiar with Python and Jupyter notebooks. Most respondents had previous experience in collaborating with others on a data science project, and working in individual Jupyter notebooks with version control tools was the most popular setting for collaboration. Team members constantly discuss and keep everyone informed about the progress of the project, as well as maintain shared notes. Although Google Colab was relatively new and was not be used by many respondents, several respondents mentioned Google Colab as an option for collaborative editing. With this more holistic understanding of collaboration among data scientists, we decided to narrow our focus and probe into the differences afforded by traditional collaboration settings, where team members work on individual Jupyter notebooks and update each others' work asynchronously, and the emerging real-time synchronized editing collaboration setting, where team members work closely together on a single Jupyter notebook with shared edits.

6 STUDY 2: OBSERVATIONAL STUDY ON REAL-TIME COLLABORATIVE DATA SCIENCE

We conducted an observational study with 24 data scientists working remotely in pairs to solve a predictive modeling problem. We tested two conditions, with users working on either individual

PID	GID	Country	Occupation	Major
P01 (M)	S1	U.S.	Student	Master in Information Science
P02 (F)	S1	U.S.	Student	Bachelor in Statistics
P03 (M)	N1	U.S.	Data Scientist	Master in Data Science
P04 (M)	N1	U.S.	Student	Master in Computer Science
P05 (F)	N2	Canada	Business Analyst	Master in Management
P06 (M)	N2	Pakistan	Software Engineer	Bachelor in Computer Science
P07 (M)	N3	India	Student	Bachelor in Information Technology
P08 (M)	N3	India	Student	Bachelor in Computer Science
P09 (M)	N4	U.S.	Student	Bachelor in Computer Science
P10 (M)	N4	Brazil	Data Scientist	Bachelor in Computer Science
P11 (M)	S2	Canada	Student	Master in Computer Science
P12 (M)	S2	Canada	Student	Bachelor in Computer Science
P13 (M)	S3	Canada	Student	Bachelor in Computer Science
P14 (M)	S3	Canada	Student	Bachelor in Computer Science
P15 (M)	S4	U.S.	Student	Bachelor in Economics
P16 (M)	S4	U.S.	Student	Master in Information Science
P17 (M)	S5	China	Software Engineer	Bachelor in Computer Science
P18 (F)	S5	China	Student	Ph.D. in Computer Science
P19 (M)	S6	U.S.	Data Scientist	Master in Computer Science
P20 (F)	S6	U.S.	Business Analyst	Master in Business
P21 (M)	N5	India	Student	Bachelor in Computer Science
P22 (M)	N5	India	Student	Master in Computer Science
P23 (M)	N6	India	Student	Master in Computer Science
P24 (M)	N6	India	Student	Master in Business

Table 3. Demographics of Participants in Study 2

notebooks or notebooks that enable synchronous editing in a collaborative setting. Our goal was to examine how collaboration styles varied between the two conditions, and to gain empirical insights on the benefits and trade-offs for each setting.

Groups chose from two communication mechanisms that were commonly used for collaboration: Slack for text-based messaging and Google Hangouts for video-based communication. In pilot studies, we found it difficult to control the communication mechanism due to technical limitations (e.g. participant microphone or network issues). Thus at the beginning of the study, we asked individual groups to decide which communication mechanisms they wanted to use throughout the study.

6.1 Participants and Task

We recruited participants who had a sufficiently substantial background in data science: (1) be familiar with Python and Jupyter notebooks, (2) have formal training in data science, and (3) have experience with predictive modeling. For reference, we provided an example predictive modeling task to potential participants for evaluating whether to opt-in to the study. We reached out to 12 initial participants from respondents to our first survey who indicated a willingness to participate in future research, and used them as seeds to recruit the rest of participants through snowball sampling. Participants (4 females, 20 males, average age = 24.62) all had basic knowledge in data

science related fields (e.g., computer science, business analytics, statistics, economics), as listed in Table 3. Seven participants currently worked as data scientists and analysts, and most (23/24) have collaborated with others on a data science project before.

Participants were randomly assigned to pairs and were instructed to work collaboratively on a predictive modeling problem. The task given was to predict housing sale prices using 80 features (e.g., lot size, type of road access, original construction date). The task involved features not relevant to the prediction, outlier records, and missing values. Pairs were asked to to develop their own strategies to judge the importance of features, handle pre-processing, and apply predictive models for sales prices. This task was based on a beginner-level competition on Kaggle[14], and indirect questions were asked in the recruitment process to ensure that participants had not worked on the same task previously. We chose the predictive modeling problem since it captures the majority activities in the data science pipeline (e.g., exploratory data analysis, data cleaning, modeling, and evaluation). In addition, the problem is open-ended, which left space for groups to try advanced models and improve the prediction result.

6.2 Apparatus

Participants joined the study remotely and their browser screens, webcams, and microphones were recorded by the research team with consent. Groups could choose to use Slack for sending text, images, or code snippets, or Google Hangouts for video calling. A JupyterHub[15] instance was made available so that participants could access the computational environment and run their notebooks from the cloud. Groups were assigned to work in either the shared condition or non-shared condition as described below.

6.2.1 Non-Shared Condition. In the non-shared condition, participants logged on to the JupyterHub platform and worked on individual notebooks. They were allowed to exchange the notebook file, set up a git repository, or send code snippets through chat or any other tool.

6.2.2 Shared Condition. In the shared condition, a Jupyter notebook collaboration extension was enabled on the server to support synchronous editing in notebooks. Participants logged on to the JupyterHub platform, created a shared notebook and invited their teammates to join the notebook.

6.3 Collaboration Extension

The collaboration extension allows code to be executed on a single interpreter while the output and runtime variables are shared among collaborators (Figure 1).

6.3.1 Workflow. To demonstrate the workflow of the extension, consider a data scientist Bob is working with his colleague Alice remotely on an exploratory data analysis problem. The extension allows Bob to enable the collaborative editing mode on the existing notebook, and share the session code with Alice. Alice then joins the notebook using the session code and edits the same notebook with Bob. The notebook updates changes in real-time, and shares the cursor positions and selections between Alice and Bob.

6.3.2 Implementation. We use Operational Transformations (OTs) to handle real-time information exchange among notebooks in the shared session, where OT is a widely-used technology for supporting consistency maintenance and concurrency control in real-time groupware systems (e.g., Google Docs). There are two types of synchronization strategies in our extension. For operations that do not involve code interpreter (e.g., editing code, adding cells, deleting cells), we use a

[14] https://www.kaggle.com/c/house-prices-advanced-regression-techniques
[15] https://jupyter.org/hub

decentralized model to update all notebooks in the shared session with recent edits. For operations that involve with code interpreter (e.g., execution a code cell, rendering a markdown cell), we first send the operation to a host notebook. We then execute the cell on the host's interpreter and update clients with output and runtime variables after the interpreter finishes execution. The extension is implemented by a Node.js web server that connects to a Postgres database, and a web-based client that implemented as a Jupyter Extension.

6.4 Study Procedure

The observational study consisted of four sessions, each of which lasted an hour. We gave participants goals for every session: to better understand data (session 1), to clean the data (session 2), to create a basic predictive model for the data (session 3), and to create a more advanced prediction model to further improve the results (session 4). Although we encouraged participants to meet the goals we set, we did not enforce these goals or any particular task ordering. Before the first session, groups were given 15 minute orientation to become familiar with each other and the study setting. We provided participants with written instructions about the study task. Groups were allowed to use external resources throughout the study.

At the end of each session, participants were asked to fill out a post-session questionnaire wherein they were asked to describe the exploration progress, their own contributions, and any difficulties they encountered in that session. Participants were also asked to evaluate a set of statements on a seven-point Likert scale. There are three themes in the statements: *communication* (e.g., being able to follow the conversation), *awareness of others* (e.g., being aware of what issues teammates are struggling with), and *group maintenance* (e.g., enjoying working with others). Lastly, the first two groups who completed the study were asked about their feedback on study settings and instructions.

After the final session, groups submitted a final prediction result, together with a merged notebook report that explained the exploration process. To motivate participants, we rewarded each participant a $40 base incentive. The group with the best prediction result was awarded an additional $10 for each member.

In addition, we conducted one-on-one post-task interviews to investigate how participants perceived the study. We first asked them to reflect on their collaboration experience (e.g., what was their strategy for coordinating work). Next we used the critical incident technique [17] to investigate if there was a situation where they could not follow the conversation, they did not feel aware of others' work, or they disturbed others' work. Participants were then prompted to think about how these incidents could be addressed if they could change the features of the tool.

6.5 Data Analysis

We used an inductive analysis approach with other methods (e.g., affinity diagramming, memoing) from grounded theory [10] to analyze the data. Screen recordings, open-ended questions from the post-session survey, and interview transcripts were first observed by the lead author to identify: (1) common collaboration patterns, (2) activities that are involved in the exploration, and (3) any challenges that were faced during collaboration. Using an open-coding approach, we first created a coding scheme based on initial observations. Four coders independently coded two samples to refine the coding scheme. We then discussed and used affinity diagramming to synthesize emerging themes. Next, we went through several iterations to check another two samples individually. The purpose of this step was to confirm the legitimacy of the coding scheme and to check the inter-rater reliability. After several iterations, four coders reached a suitable level of agreement (Fleiss's kappa, $\kappa = 0.71$).

Group ID	Collaboration Style	Notebook Ratio	Message Ratio	Error Score
N1	Competitive Authoring	88%	61%	0.21
N2	Divide and Conquer (datasets)	71%	54%	0.36
N3	Competitive Authoring	84%	61%	0.25
N4	Competitive Authoring	69%	55%	0.48
N5	Divide and Conquer (datasets)	68%	59%	0.23
N6	Competitive Authoring	91%	51%	0.12
S1	Divide and Conquer (tasks)	58%	61%	0.13
S2	Single Authoring	94%	N/A	0.15
S3	Divide and Conquer (tasks)	71%	58%	0.21
S4	Divide and Conquer (tasks)	67%	N/A	0.16
S5	Single Authoring	83%	75%	0.23
S6	Pair Authoring	86%	N/A	0.16

Table 4. Collaboration Styles. The notebook ratio is the percentage of cells that one member contributes in the final notebook. The message ratio is the percentage of messages that one member sends in total Slack messages; N/A means the team uses video-based communication (Google Hangouts). The error score is calculated using Root Mean Square Error (RMSE). Lower error scores are better.

7 KEY FINDINGS FROM STUDY 2

We first present the results of the common collaboration styles that emerged in the two collaboration settings. Based on the framework of collaborative writing developed by Posner and Baecker [60], we propose four collaboration styles. Then, we present an analysis of our comparison of communication styles, finding that working in the synchronized notebook can reduce the communication costs by establishing a shared context. We also report the differences in performance (e.g., final prediction scores, number of alternative models, notebook lengths) between the two collaboration settings. Lastly, we present the challenges that participants faced in using real-time collaborative editing features.

7.1 Collaboration and Communication Styles

Based on what we observed in our study and extending the framework developed by Posner and Baecker [60] for collaborative writing styles, we refine and propose four collaboration styles for data science tasks based on team members' contributions in ideation and implementation.

- **Single Authoring**: The single authoring style is extended from the *single writer* strategy from Posner's framework. In this style, one team member contributed the majority of ideas and did the majority of implementation, while the other did not provide substantial contributions.
- **Pair Authoring**: The pair authoring style is extended from the *scribe* strategy from Posner's framework where one team member completed the majority of the implementation while the other contributed ideas, engaged in discussions and reviewed the results. We chose the term pair authoring because it is as analogous to the collaboration style in pair programming where one person writes the code and the other reviews the code. This is different from pair programming where two programmers frequently switch roles: here, the individuals engaged in pair authoring stick to their roles from the beginning to the end.
- **Divide and Conquer**: The divide and conquer style is a combination of the *separate writers* strategy and *joint writing* strategy from Posner's framework. Here, participants divided the

task into subgoals and explored the subgoals independently. We observed two types of divide and conquer strategies — **(1) dividing datasets**, where participants split the dataset into half and explore in parallel using the same technique, **(2) dividing tasks**, where participants split the task and work on different parts in parallel.

- **Competitive Authoring**: We proposed this collaboration style based on our observation of team members going through an idea together and competitively implementing it. It is different from divide and conquer that team members wrote the code for the same purpose and reached the consensus to use the code by whoever finished first. There is no equivalent strategy in Posner's framework.

We then used these definitions to code each group's collaboration style. We analyzed the notebook contribution ratio (the percentage of cells that one member contributes in the final notebook) and the message ratio (the percentage of messages that one member sends out of the total number of Slack messages) with the code and notes from screen recordings to inform our judgment. We report how groups adopted different collaboration styles using a collaborative editing notebook (shared condition) and using individual notebooks (non-shared condition) in Table 4.

7.1.1 Single Authoring Style. Two groups adopted the single authoring style when working in the synchronized (shared condition) notebook. We observed an unbalanced contribution in their final notebooks. Participants who contributed less reported that without strategic coordination and communication, they did not have a task that fit their level of expertise and ended up walking through the same task with their teammate. One explanation that arose is that the participants might feel pressured and off-topic, not knowing how to engage in the task:

> ... My teammate is better and faster in doing the task. Sometimes I know he is trying an idea, but it may take me a while to figure it out and he just jumped to the next task... I wish I could have a separate window to try the code in my own pace... (P12 from S2)

7.1.2 Pair Authoring Style. One group used the pair authoring style when working in the same notebook. Two members agreed that one person was in charge of implementation while the other helped with ideation and finding documentation. As opposed to single authoring, where one group member contributes the majority of code and ideas, in pair authoring, both group members felt engaged in decision making. They further explained why they found pair authoring useful using the pair programming metaphor of a *driver* who writes the code, and a *navigator* who reviews the code:

> ... There is a dynamic in pair programming called "Driving and Navigating". The idea is that two people on one keyboard is difficult to do well, so one should primarily be leading... (P19, the main driver from S6)

> ... Data science is not just about writing code, you know. My role is as important as my teammate. I can think about the big picture while my teammate is working on the details in Python. It is not necessary for me to step into his code... (P20, the main navigator from S6)

7.1.3 Divide and Conquer. We observed the pattern of "divide and conquer" from five groups, with three groups from the shared condition and two from the non-shared condition. Participants used different strategies to decide how to divide the task. Some groups planned ahead to discuss the goal of the session and used that to guide how to divide the task. These groups often decided whom to assign the sub-tasks to based on group members' skills. For example, participants in group S3 used markdown cells to list things they wanted to explore and put group members' name aside to track the tasks. Other groups did not plan ahead and used ad-hoc planning, keeping each other updated

about what they were doing so that the other group member could find a new task to work on. For example, P2 in group S1 said to P1, "OK, while you fix the stuff, I'll create one hot encoding for categorical variables". However, participants explained their concerns that not knowing enough about another person's skill set made it difficult to split the work:

> ... I wish I had a more personal relationship with my fellow research participant so I didn't feel weird about judging who should do what based on skill... (P13 from S3)

> ... Basically, I don't like having to make a judgment about the ability of my partner without having the benefit of knowing our strengths and desires relative to each other. It makes me self-conscious that I'm not listening to my partner enough... (P16 from S4)

In addition, we identified two types of divide and conquer strategies: dividing *datasets* and dividing *tasks* based on alternative solutions. Both groups from the non-shared condition split the dataset and had each member walked through half of the features. This strategy can boost efficiency in exploratory data analysis given that there are roughly 80 features in the dataset. It can help two members to establish common ground in the general task. However, this might result in duplicated implementation. For example, both group members wrote their own code to plot the distribution of certain features. None of the groups in the shared condition used this strategy. Instead, they divided the task by alternative solutions. For example, in the data modeling stages, participants came up with different models (e.g., decision trees, elastic nets) and assigned each person the task to explore one model.

7.1.4 Competitive Authoring. Non-shared groups often used competitive authoring, where group members would competitively implement the same idea. They would either choose to copy the code from whomever finishes the implementation first or choose to keep their own version of implementation. For example, two group members were both working on the implementation of linear regression, while one member got their code working faster than the other member. The other member would agree to copy the code to their own notebook and move on to the next task. There can be an unbalanced distribution in the final work and working efficiency can decrease because one member is always writing "unnecessary" code. However, from the individuals' perspective, they sometimes felt it necessary to explore the code on their own:

> ... Each would do the same tasks and share the insights ... I don't think it is wasting time. When we both wrote the code, at least we were on the same page. When he shared his code, I can have a better understanding of what was going on... (P24 from N6)

7.2 Communication Channels

Web conferencing is perceived to have high communication bandwidth with synchronous and immediate information exchange, whereas instant messaging and chat tools have a lower communication bandwidth and support both synchronous and asynchronous information exchange [67]. The fact that participants chose to use Slack rather than Hangouts in the non-shared condition (without prompting from the study coordinators) indicates that the affordances of Slack with a variety of media formats, as well as the ability to represent conversation artifacts (e.g. source code) in a manner which fit their needs. Data scientists need to constantly share intermediate code or outcomes with each other. Working in the shared notebook reduces the communication costs by providing a shared context for discussion. For example, a participant from the non-shared condition explained that they preferred Slack messaging over Hangouts, and they wished to have the shared notebook for better communication:

> ... would be much nicer to work on the same notebook rather than copying code in Slack but worked much better than Hangouts... (P7 from N3)

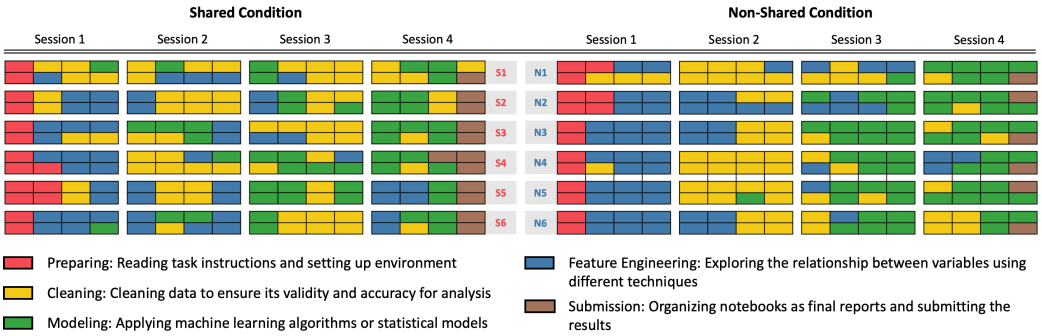

Fig. 2. Overview of how participants iteratively explore the house price prediction task in 15-minute intervals. Participants in the shared condition tended to switch between phases more frequently. The initial attempts at modeling occurred earlier in the shared condition.

In addition, we examined the types of messages participants sent in each condition. Overall, we found that participants in the non-shared condition used Slack more often to send files, code snippets, and output (e.g., data pieces, error messages, visualization, screenshots of the notebook). Every group in the non-shared condition exchanged their notebooks at least once during the study session. Groups in the non-shared condition also sent more execution output such as data pieces and images in the Slack channel. Moreover, they sent at least four times as many code snippets during the study, while groups in the shared condition rarely did so. This result suggests that working in the same notebook can reduce the communication costs by establishing a shared context.

7.3 Working Across Phases

As we describe in section 6.4, we gave participants recommended goals for each of the four sessions but did not require that they met these goals or worked in any particular order. In order to understand participants' exploration patterns and how they differ across both conditions, we segmented the screen recordings into 15-minute intervals and categorized participants' activities. We found that participants typically did any given activity for a minimum of 10–15 minutes so 15 minutes was an appropriate level of granularity. For each 15-minute interval, we categorized their activity into one of the common data science phases described by O'Neil and Schutt [55][16]:

- Preparing: Reading task instructions and setting up the environment
- Cleaning: Cleaning data to ensure its validity and accuracy for analysis
- Modeling: Applying machine learning algorithms or statistical models for prediction
- Feature Engineering: Exploring the relationship between variables using different techniques
- Submission: Organizing notebooks as final reports and submitting the results

Figure 2 shows the activity classifications for every group across every session. Participants did not perform tasks in sequence — they frequently backtracked and switched between different phases of analysis as necessary. Participants in the shared condition tended to switch between phases more frequently. We calculated how many times participants switched between phases (when the adjacent tiles have different colors in Figure 2) and ran a two-sample T-test on the result. We found that participants in the shared condition switched more frequently (avg=8.58, $p=0.000043$) than participants in the non-shared condition (avg=5.83). This result indicated that

[16]We omitted the "data munging" and "data collection" phases because participants were given an existing dataset and did not need to collect raw data themselves.

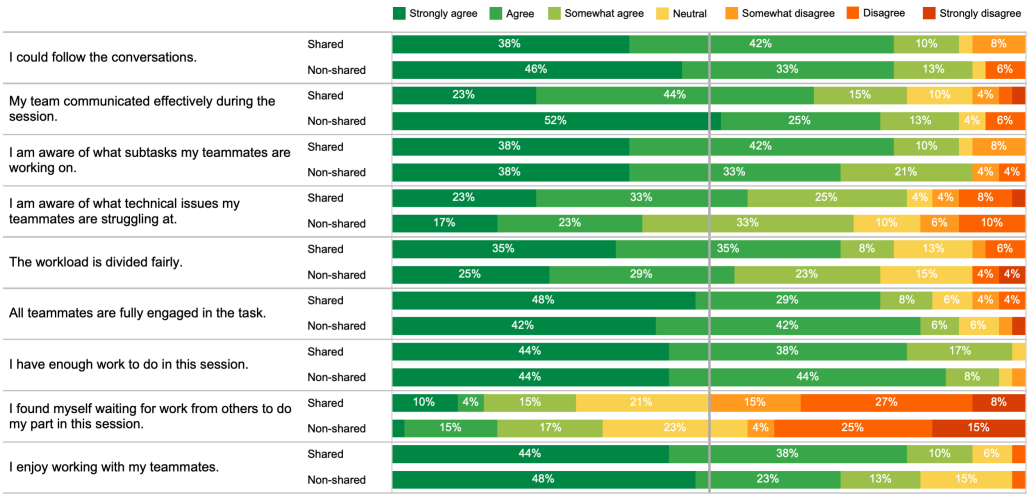

Fig. 3. Post-task questionnaire results for participants in both conditions.

working on the same notebook provides collaborators with convenience to branch through tasks. Perhaps as a result of this convenience, the initial attempts at modeling (green squares in Figure 2) occurred earlier in the shared condition than in the non-shared condition. As Figure 2 also shows, in every group in the non-shared condition, one participant bore the responsibility of organizing the notebooks for submission (brown squares in Figure 2). By contrast, nearly every participant in the shared condition helped organize the notebook and results for the final report.

7.4 Overall Effectiveness — Accuracy and Questionnaire Results

We found an improvement in the exploration process across two conditions. As shown in Table 5, we run a two sample T-test to compare the outcomes from the final notebooks and prediction results across two conditions. Groups working in the same notebook (avg error score[17] = 0.17) achieved a better prediction result compared to groups working in individual notebooks (avg error score=0.27, p=0.09). Groups working in the same notebook explored more alternative models (avg=6.17) compared to groups working in individual notebooks (avg=3.00, p=0.05). In addition, we compared the post-session survey results across all sessions, as illustrated in Figure 3. Participants' rating of their enjoyment working with teammates was significantly improved when working in the shared condition after the first session (p=0.04). We also compared the total lines of code in the final notebook and found a significant difference in the shared condition (avg=186.67) and the non-shared condition (avg=90.33, p=0.04). This result suggests that working in the same notebook encourages groups to explore more solutions and leads to a better result. For example, P2 noted:

> ... Overall, I think the tool is amazing! This tech can really increase productivity in data science teams!... (P2 from S1)

When we compared the ratio of annotation cells and total cells from the final submission, we did not see differences across two conditions. Moreover, the average ratio of annotation cells to total cells was lower in the shared condition (avg=0.19) compared to the non-shared condition

[17]The error score is calculated using Root Mean Square Error (RMSE)

		Shared	Non-shared
Error Score	$\overline{x}$	0.17	0.27
	σ	0.04	0.13
Number of Models	$\overline{x}$	6.17	3.00
	σ	2.99	2.10
Lines in the Notebook	$\overline{x}$	186.67	90.33
	σ	82.10	64.50
Percentage of Annotation Cells	$\overline{x}$	0.19	0.20
	σ	0.11	0.13

Table 5. Comparing the outcomes from prediction results and final notebooks (mean: $\overline{x}$, standard deviation: σ). Working in the same notebook encourages groups to explore more solutions and leads to a better result.

(avg=0.20). This result indicates that when working in groups, participants would not pay extra attention to add annotations into the notebook compared to working in a private notebook.

7.5 Challenges in Using the Collaborative Notebooks

Despite all benefits that collaborative editing features offer with respect to sharing context and improving productivity, we discovered several challenges in using the collaborative notebooks. We present the key findings below.

7.5.1 Interference with Each Other. Over half of the participants in the shared condition (7/12) raised concerns about interference with each other in the post-task interviews. Participants would take ownership of the code cells they created. They would expect others not to edit "their" cells. Some participants even took actions such as inserting blank cells between each others' editing areas to prevent interference. Participants also reported that different naming styles could cause trouble, especially changing a variable name halfway through the exploration:

> ... When using Jupyter notebooks together, it's hard to keep track of variable names. Everyone might use a different name and may cause issues. For example, my teammate used *train_df* as name, and later changed it to something else, but I wanted him to keep using the original name... (P2 from S1)

In addition, we observed that during exploration, some participants directly modified the shared data frame (Figure 1.3) without making copies or notifying their teammates. For example, P14 from S3 spent a long time debugging the error score for the basic linear regression model and finally realized that his teammate had transformed the scale of the shared data frame for other purposes.

7.5.2 Lack of Awareness. Although the collaborative Jupyter notebook shares cursor positions and selections with collaborators, participants reported that this mechanism was not enough to understand what their teammates' activities. First, participants would have to scroll the notebook frequently to check their teammates' edits, which can be difficult when the notebook grows rapidly during quick exploration. Second, if the participant only wants to know the high-level task his teammate is working on, it takes time for him to read and understand the code when it is not well annotated. Lastly, the cursor information may not reflect their teammates' activities, especially when doing data science needs reasoning and decision making. For example, participants reported:

> ... I want it to be easier for my research partner to show me what they're working on. I felt it was difficult to do something quickly on the side without affecting what my partner was working on... (P17 from S5)

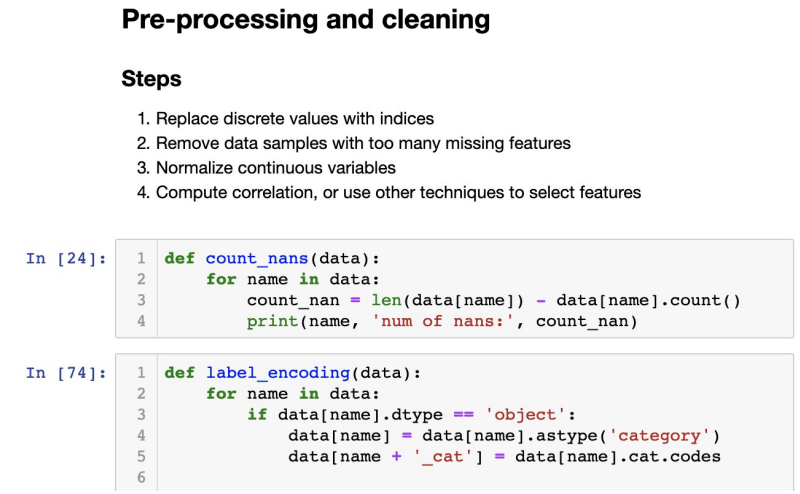

Fig. 4. Group S3 coordinated the work well by planning subgoals in the notebook

> ... It's hard to keep track of what my teammate is doing while he's not writing code on notebook because I don't see him physically... (P12 from S2)

7.5.3 Problems with the Linear Structure. Cells are displayed linearly in Jupyter notebooks but the actual execution order may not follow a linear structure. As mentioned, participants may divide notebook cells into regions based on a sense of ownership. Participants may also iterate and jump through the code cells for ad-hoc divide and conquer. Our analysis of the final notebooks indicates that although the amount of code grew significantly faster in shared notebooks than in individual notebooks, the percentage of annotation cells in the shared notebooks was smaller than in individual notebooks. Such ill-organized cells made it difficult for collaborators to navigate the notebook. In fact, we were not able to run the cells sequentially to reproduce the results in two of the six notebooks submitted by groups in the shared condition. One participant suggested separating the preliminary exploration with the main notebook:

> ... I want to distribute and segment work more easily, but notebooks fundamentally struggle with this due to the restrictive UI. The cells are stacked top to bottom. There is no concept of a "scratch cell" or "main script". This makes it difficult to say, "here is the main section of code, and we're writing exploratory code on the side to prototype improvements". I appreciate that kind of paradigm ... (P19 from S6)

Participants also referred to different ways to support non-linear structures. For example, one participant proposed to track the execution order in a file which is similar to the example of "Makefiles" in C programming. Another participant suggested to break down the structure of notebook cells into parallel branches. He offered an example:

> ... I'd love to see cells as nodes and edges to their dependencies. Imagine LR (logistic regression) and RF (random forest) models. LR needs data to be scaled and preprocessed. RF does not need it. You could set up cells with explicit dependencies to the appropriate preprocessing steps. You can imagine this as data flowing through containers of code... (P15 from S4)

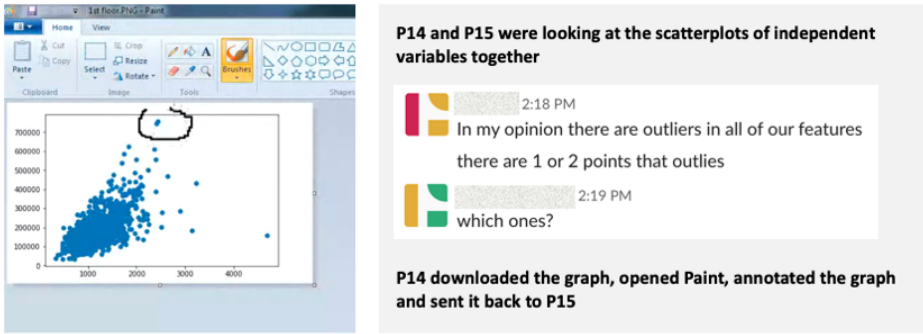

Fig. 5. An example of participants manually annotating a graph for discussion: P14 circled the outlier point in a scatter plot using MS Paint and sent it back to P15 using Slack.

7.5.4 Privacy Concerns. Two groups in the shared condition wrote using the single authoring style—one participant in both groups contributed the majority of ideas and implementation. Both participants who shied away from the task mentioned that they felt pressured to edit the code cells despite knowing that their teammates were better than them. These participants also mentioned that they only wanted to share the code when it was done. Such concern was also reported by an experienced participant:

> ... At the beginning I was hesitant to edit the notebook. It was much better later on because I knew he was busy with something else so that he wouldn't pay attention to my code... (P2 from S1)

These concerns are analagous to the finding of Wang et al. that some writers are concerned with being judged or distracted in the context of collaborative writing [69].

7.5.5 Lack of Strategic Coordination. Lack of strategic coordination also resulted in unbalanced contributions. After the last session, participants reported a lower agreement on "I have enough work to do in this session" in the shared condition than in the non-shared condition (p=0.08, Figure 3). The participant who contributed more in the single authoring group acknowledged that the work was not divided equally because they did not plan ahead:

> ... I feel I am not splitting work well enough. I was thinking about how to get the work done and just tried the ideas on myself.... (if doing it again) I will probably ask him to help with data cleaning... (P11 from S2)

Another group (S3) demonstrated successful coordination. They listed subgoals in the notebook and assigned tasks based on preferences (Figure 4). This planning and negotiation also helped them to divide the code cell regions with nested headings. While some groups also discussed the subgoals verbally or in Slack, we did not see a clear notebook structure. Retrieving such information can be useful to automatically help collaborators to manage their shared notebook. We will elaborate on this in the discussion.

7.5.6 Contextual Chatting. As we describe in section 7.2, participants in the shared notebook condition sent fewer code snippets, images, execution results, and screenshots on average — likely because all of these elements were automatically synchronized across all team members, which eliminates the need for users to share them manually. As a result, we expected participants in the shared condition to believe their teams communicated more effectively than participants in the non-shared condition. However, we found the opposite—participants in the shared condition

reported that they felt their groups communicated worse than participants in the non-shared condition (as measured by participants' agreement with "my team communicated effectively during the session" after session 2, p=0.08, Figure 3). One possible reason for this is that participants in the shared condition switched between phases more frequently (as Figure 2 illustrates), which could make communication more challenging.

In the post-task interview, we specifically asked about the difficulties participants had with communication. Most participants (9/12) mentioned that constantly switching between Slack and Jupyter notebook was distracting. Several participants did not enable notifications of Slack messages and were not able to respond to their teammates immediately. Participants also felt the need to refer to some output (e.g., a specific column of a table, an area in visualization) during discussion but current tools do not support deictic references. Figure 5 shows an example of participants manually annotating a visualization for discussion: P15 was not clear where the outlier points were in the scatter plot so P14 had to download the graph from notebook, opened Microsoft Paint to circle the point and sent it back to P15 through Slack.

8 DISCUSSION

Our main findings indicate that synchronous editing helps data scientists maintain a shared understanding while reducing communication costs, thus improving the overall efficiency of collaboration. However, current synchronous editing features can be challenging to use and require collaborators to be strategic with respect to coordination. Below, we contextualize our findings with existing frameworks and findings on collaborative editing in other contexts, and highlight the needs for a human-centered approach to study different collaboration scenarios in data science. We discuss future directions to improve the current design of synchronous notebook editing features to better support teamwork among data scientists.

8.1 Extending Our Understanding of Collaborative Editing Across Contexts

Some of the challenges we identified with real-time notebook editing are related to prior studies on collaborative writing systems and collaborative coding systems. For example, the privacy concern of being watched by others while working has been observed in other contexts [11, 69]. This issue may be more pronounced for collaborative data science, because data science requires a large amount of experimentation with code and collaborators may hold different programming backgrounds and domain knowledge, something noted by others (e.g. [40]) and observed by ourselves in this work. In addition, there is an opportunity for future human-centered studies to explore different roles in collaborative data science. Additional surveys and ethnographic work can aid the understanding of the whole spectrum of human-centered data science work. It is also worth exploring novel designs in this space. For example, folding code blocks [63] and implementation details might allow domain experts (e.g., marketing specialists) to be able to understand and experiment with model parameters in a notebook. Another challenge for data scientists working on the same notebook is interference of work, which is also a challenge for collaborative code editors [20]. However, it is less likely for compilation errors to impede collaboration in shared notebooks than in collaborative code editors, as the design of Jupyter notebooks allows users to run individual cells. If one user writes a code cell with compilation error, other users can simply skip this cell and run other codes. We found the interference happened in shared notebooks mainly due to conflicts in shared data frames.

In addition to findings consistent with other studies of collaborative editing, collaborative editing in computational notebooks has its unique aspects. The mixed form of code and other types of media has distinguished computational notebooks from textual documents and pure code scripts. For instance, collaborative writing systems usually share static text synchronously whereas programming typically share their codebase through asynchronous Version Control Systems

(VCSs)[18]. In shared notebooks, however, it remains unknown what the level of synchronicity should be (e.g., sharing static text and code, sharing the output, sharing the code interpreter), in part because of the emphasis on the sensemaking and experimentation processes.

Further, whereas it is common for programmers to segment code into modules based on their functions and eventually work on different files, data scientists rarely split their work into multiple notebooks. Thus, integrating version control locally [35] can be one potential solution to help collaborators track each others' edits.

8.2 Opportunities and Challenges of Collaboration in Computational Notebooks

Despite all the benefits of working in shared notebooks—encouraging more exploration and reducing communication costs—it is not easy to judge whether working in collaborative notebooks as currently designed is better than working on individual notebooks. For example, data science learners may find it more useful to work on a private notebook and to explore a task privately first before discussing the results with their collaborators. Reflecting on the context of collaborative writing, the common collaborative editing features for writing include tracking changes for review, adding comments, adding access control for the whole document. Tools like Google Docs are designed to support more than real-time editing, and studies have found that users rarely edit the same piece simultaneously in practice [11]. How teams choose to use collaborative writing tools will depend on their goals and work preferences. For example, the "track changes" and comments features may be more useful when collaborators engage in the same document asynchronously. Thus, designers should take a user-centered design approach and reflect on different purposes of collaboration when extending the collaborative editing features to the context of notebook editing.

Our observational study explored one specific scenario where data scientists who did not know each other worked simultaneously over four hours to solve a predictive modeling problem. It may not be representative of all of all data science collaboration scenarios. Nonetheless, it is important as a first step to understand the challenges in current collaborative notebook editing features. We believe that some challenges can transfer to other collaboration scenarios. For instance, when collaborators edit the same notebook in a different time, they may still want more awareness information on what their partner is working on. Future work should explore how to generalize the design to serve the needs for various collaboration scenarios in real-world data science practice.

8.3 Design Implications

Studies have explored approaches to improve the infrastructure of computational notebooks for individual authoring and sharing (e.g., enabling content folding [62] and tracking exploration history [38]). Our study explicitly examined the benefits and challenges for multiple users to edit the same notebook collaboratively. The results suggest to the needs for a better collaborative notebook infrastructure. Below we discuss several design opportunities.

8.3.1 Improve Awareness of Collaborators' Activity. Our current synchronous notebook editor design tracks and displays the locations of cursors of collaborators. However, sharing cursor information did not seem to be enough for users to perceive changes from others given a shared interpreter (back-end python process) state. Further, it is challenging to track the shared cursor when the notebook gets long. It would be valuable to explore what information requires high awareness and how to present the awareness information—particularly for large and complex notebooks. For example, we may infer the context of the code that one person is working on from the nearest narrative text or headings and share the heading with other users. We may also

[18]Code editors that synchronize text content in real-time are more useful for learning and tutoring purposes where there are few lines of code and the cost of potential conflicts are minimal.

broadcast important changes made to certain cells (e.g., cells that initiate variable names, or import libraries), or more explicitly share changes to the shared interpreter state..

8.3.2 Provide Access Control. We observed a strong tendency for participants to take ownership of the cells they worked on. Although the notebook and execution environment were shared, participants did not want others to edit their cells without permission. On the other hand, some participants did not want others to see their edits on a cell until they deemed it as "finished". These findings suggest that access control mechanisms may be appropriate to integrate into the collaborative notebook infrastructure. We propose that there are at least two types of access control, which we called either *editing control* or *visibility control*. For editing control, we could imagine integrating the local versioning design [38] to protect a cell so that collaborators can submit their edits for approving. For visibility control, users could instead choose to disable the real-time synchronicity for certain cells or choose to fold the implementation details (e.g., [62]), thus hiding the work until it was ready.

8.3.3 Enable Discussions within Notebooks. Frequent communication is important for data scientists to stay updated on progress, reason about decisions, and coordinate work. Participants found it difficult to use third-party instant messaging tools because they had to constantly switch between applications. In addition, we observed that participants referred back to the shared notebook content often. This suggests that there is value in exploring the design of an in-notebook chat window. One potential benefit of such design is to support deictic references to a specific part of the notebook (e.g., [56]).

Moreover, there is rich information in chat messages; users report their progress, make plans, or explain parts of code. Investigating how to utilize the chat history to help users annotate the notebook may help moderate the tension between quick exploration and clear explanation [63].

8.4 Limitations

Since we only looked at one specific scenario of collaboration, our results and design implications may not effectively represent the needs for a better collaboration tool for other collaboration scenarios in data science. For example, the type of data science problems, the expertise of collaborators, team size, the synchronicity of the collaboration, and whether a final narrative is the end goal may all affect how users perceive the synchronous editing features. In addition, we scheduled the study sessions with remote participants at their convenience, which resulted in pairs more likely from the same region. Future work should explore broader collaboration settings and broader demographics.

9 CONCLUSION

We probed into how synchronous editing in computational notebooks might change the way data scientists collaborate on a predictive modeling task. Our survey findings highlight the tools and strategies that data scientists currently used in collaboration practice. Based on the design of current synchronous editing features in computational notebooks, our empirical observation reveals that working on the same notebook results in different collaboration styles compared to working on individual notebooks. The key findings suggest that synchronous editing tools improve collaboration by helping data scientists maintain a shared context and improve work efficiency. However, the current real-time collaborative editing features may lead to several problems (e.g., interference with each others' work, unbalanced contributions). The challenges in using the current real-time collaboration features suggest that we need better collaborative editing features for computational notebooks. We discuss how our results extend prior work on collaborative editing and how the HCI community can play a vital role in broadening the understanding of collaborative data science with a human-centered approach. Finally, we propose design implications to enhance

synchronous editing in computational notebooks and to improve collaboration among data science workers.

10 ACKNOWLEDGEMENTS

We thank John Penington for his work creating a proof of concept implementation of the shared Jupyter notebook. We also thank Natalie Gross, Jamie Neumann, and Rebecca Parada for their help with coding and analyzing data from our second study. We thank all of our participants across both studies and our reviewers for their valuable feedback. We also thank the Michigan Institute for Data SNaticience (MIDAS). This material is based upon work supported by the National Science Foundation under Grant No IIS 1755908.

REFERENCES

[1] 2018. LinkedIn Workforce Report | United States | August 2018. https://economicgraph.linkedin.com/resources/linkedin-workforce-report-august-2018

[2] Ronald M Baecker, Dimitrios Nastos, Ilona R Posner, and Kelly L Mawby. 1995. The user-centred iterative design of collaborative writing software. In *Readings in Human–Computer Interaction*. Elsevier, 775–782.

[3] Andrew Begel. 2008. Effecting Change: Coordination in Large-scale Software Development. In *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '08)*. ACM, New York, NY, USA, 17–20. https://doi.org/10.1145/1370114.1370119

[4] Jeremy Birnholtz, Stephanie Steinhardt, and Antonella Pavese. 2013. Write Here, Write Now!: An Experimental Study of Group Maintenance in Collaborative Writing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 961–970. https://doi.org/10.1145/2470654.2466123

[5] Netflix Technology Blog. 2018. Beyond Interactive: Notebook Innovation at Netflix. https://medium.com/netflix-techblog/notebook-innovation-591ee3221233

[6] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. 2008. Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies* 66, 7 (2008), 519 – 529. https://doi.org/10.1016/j.ijhcs.2007.03.005 Collaborative and social aspects of software development.

[7] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S. Lasecki, and Steve Oney. 2017. Codeon: On-Demand Software Development Assistance. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6220–6231. https://doi.org/10.1145/3025453.3025972

[8] Yan Chen, Steve Oney, and Walter S. Lasecki. 2016. Towards Providing On-Demand Expert Support for Software Developers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 3192–3203. https://doi.org/10.1145/2858036.2858512

[9] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 977–989. https://doi.org/10.1145/3242587.3242600

[10] Juliet Corbin and Anselm Strauss. 2008. *Basics of qualitative research: Techniques and procedures for developing grounded theory, 3rd ed.* Sage Publications, Inc. https://doi.org/10.4135/9781452230153

[11] Gabriele D'Angelo, Angelo Di Iorio, and Stefano Zacchiroli. 2018. Spacetime Characterization of Real-Time Collaborative Editing. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 41 (Nov. 2018), 19 pages. https://doi.org/10.1145/3274310

[12] Thomas H. Davenport and D. J. Patil. 2012. Data Scientist: The Sexiest Job of the 21st Century. (2012). Issue October 2012. https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century

[13] Prasun Dewan and John Riedl. 1993. Toward computer-supported concurrent software engineering. *Computer* 26, 1 (Jan 1993), 17–27. https://doi.org/10.1109/2.179149

[14] David Donoho. 2017. 50 Years of Data Science. *Journal of Computational and Graphical Statistics* 26, 4 (2017), 745–766. https://doi.org/10.1080/10618600.2017.1384734

[15] Paul Dourish and Victoria Bellotti. 1992. Awareness and Coordination in Shared Workspaces. In *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work (CSCW '92)*. ACM, New York, NY, USA, 107–114. https://doi.org/10.1145/143457.143468

[16] Hongfei Fan, Chengzheng Sun, and Haifeng Shen. 2012. ATCoPE: Any-time Collaborative Programming Environment for Seamless Integration of Real-time and Non-real-time Teamwork in Software Development. In *Proceedings of the 17th ACM International Conference on Supporting Group Work (GROUP '12)*. ACM, New York, NY, USA, 107–116. https://doi.org/10.1145/2389176.2389194

[17] John C Flanagan. 1954. The critical incident technique. *Psychological bulletin* 51, 4 (1954), 327.

[18] Gregg Stanley Foster. 1986. *Collaborative Systems and Multi-user Interfaces*. Ph.D. Dissertation. AAI8717981.

[19] Max Goldman. 2010. Test-driven Roles for Pair Programming. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 515–516. https://doi.org/10.1145/1810295.1810458

[20] Max Goldman. 2012. *Software Development with Real-time Collaborative Editing*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0829066.

[21] Max Goldman, Greg Little, and Robert C. Miller. 2011. Real-time Collaborative Coding in a Web IDE. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 155–164. https://doi.org/10.1145/2047196.2047215

[22] Philip J. Guo. 2012. *Software tools to facilitate research programming*. Ph.D. Dissertation. Stanford University Stanford, CA.

[23] Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology (UIST '15)*. ACM, New York, NY, USA, 599–608. https://doi.org/10.1145/2807442.2807469

[24] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance (TaPP'12)*. USENIX Association, 7–7. http://dl.acm.org/citation.cfm?id=2342875.2342882

[25] Carl Gutwin and Saul Greenberg. 1998. Effects of Awareness Support on Groupware Usability. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '98)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 511–518. https://doi.org/10.1145/274644.274713

[26] Carl Gutwin and Saul Greenberg. 2002. A Descriptive Framework of Workspace Awareness for Real-Time Groupware. *Computer Supported Cooperative Work (CSCW)* 11, 3 (01 Sep 2002), 411–446. https://doi.org/10.1023/A:1021271517844

[27] Caroline Haythornthwaite. 2005. Introduction: Computer-Mediated Collaborative Practices. 10, 4 (2005). https://doi.org/10.1111/j.1083-6101.2005.tb00274.x

[28] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 270, 12 pages. https://doi.org/10.1145/3290605.3300500

[29] Jeffery Heer and Maneesh Agrawala. 2007. Design Considerations for Collaborative Visual Analytics. In *2007 IEEE Symposium on Visual Analytics Science and Technology*. 171–178. https://doi.org/10.1109/VAST.2007.4389011

[30] James D. Herbsleb and Audris Mockus. 2003. Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering. *SIGSOFT Softw. Eng. Notes* 28, 5 (Sept. 2003), 138–137. https://doi.org/10.1145/949952.940091

[31] Petra Isenberg, Niklas Elmqvist, Jean Scholtz, Daniel Cernea, Kwan-Liu Ma, and Hans Hagen. 2011. Collaborative visualization: Definition, challenges, and research agenda. *Information Visualization* 10, 4 (2011), 310–326. https://doi.org/10.1177/1473871611412817 arXiv:https://doi.org/10.1177/1473871611412817

[32] Project Jupyter. 2015. Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science. https://blog.jupyter.org/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science-2b5fb94c3c58

[33] Project Jupyter. 2018. Jupyter receives the ACM Software System Award. https://blog.jupyter.org/jupyter-receives-the-acm-software-system-award-d433b0dfe3a2

[34] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. 18 (2012), 2917–2926. https://doi.org/10.1109/TVCG.2012.219

[35] Mary Beth Kery, Amber Horvath, and Brad A. Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[36] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 92, 13 pages. https://doi.org/10.1145/3290605.3300322

[37] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2017-10). 25–29. https://doi.org/10.1109/VLHCC.2017.8103446

[38] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 147–155. https://doi.org/10.1109/VLHCC.2018.8506576

[39] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 174, 11 pages. https://doi.org/10.1145/3173574.3173748

[40] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering* (2016) *(ICSE '16)*. ACM, 96–107. https://doi.org/10.1145/2884781.2884783

[41] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[42] Laura Koesten, Emilia Kacprzak, Jeni Tennison, and Elena Simperl. 2019. Collaborative Practices with Structured Data: Do Tools Support What Users Need?. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 100, 14 pages. https://doi.org/10.1145/3290605.3300330

[43] Kenneth L. Kraemer and John Leslie King. 1986. Computer-based Systems for Cooperative Work and Group Decision-making: Status of Use and Problems in Development. In *Proceedings of the 1986 ACM Conference on Computer-supported Cooperative Work (CSCW '86)*. ACM, New York, NY, USA, 353–375. https://doi.org/10.1145/637069.637115

[44] Robert Kraut, Carmen Egido, and Jolene Galegher. 1988. Patterns of Contact and Communication in Scientific Research Collaboration. In *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work (CSCW '88)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/62266.62267

[45] Sean Kross and Philip J. Guo. 2019. Practitioners Teaching Data Science in Industry and Academia: Expectations, Workflows, and Challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3290605.3300493

[46] J. Chris Lauwers and Keith A. Lantz. 1990. Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90)*. ACM, New York, NY, USA, 303–311. https://doi.org/10.1145/97243.97301

[47] Sang Won Lee, Yan Chen, Noah Klugman, Sai R. Gouravajhala, Angela Chen, and Walter S. Lasecki. 2017. Exploring Coordination Models for Ad Hoc Programming Teams. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17)*. ACM, New York, NY, USA, 2738–2745. https://doi.org/10.1145/3027063.3053268

[48] Sheng Feng Li and Andy Hopper. 1998. A framework to integrate synchronous and asynchronous collaboration. In *Proceedings Seventh IEEE International Workshop on Enabling Technologies: Infrastucture for Collaborative Enterprises (WET ICE '98) (Cat. No.98TB100253)*. 96–101. https://doi.org/10.1109/ENABL.1998.725678

[49] Paul Benjamin Lowry, Aaron Curtis, and Michelle René Lowry. 2004. Building a Taxonomy and Nomenclature of Collaborative Writing to Improve Interdisciplinary Research and Practice. *The Journal of Business Communication* 41, 1 (2004), 66–99. https://doi.org/10.1177/0021943603259363

[50] Alex Mitchell, Ilona Posner, and Ronald Baecker. 1995. Learning to Write Together Using Groupware. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 288–295. https://doi.org/10.1145/223904.223941

[51] Pascal Molli, Hala Skaf-Molli, Gérald Oster, and S. Jourdain. 2002. SAMS: synchronous, asynchronous, multi-synchronous environments. In *The 7th International Conference on Computer Supported Cooperative Work in Design*. 80–84. https://doi.org/10.1109/CSCWD.2002.1047653

[52] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How Data Science Workers Work with Data: Discovery, Capture, Curation, Design, Creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 126, 15 pages. https://doi.org/10.1145/3290605.3300356

[53] Judith S. Olson, Gary M. Olson, Marianne Storrøsten, and Mark Carter. 1993. Groupwork Close Up: A Comparison of the Group Design Process with and Without a Simple Group Editor. *ACM Trans. Inf. Syst.* 11, 4 (Oct. 1993), 321–348. https://doi.org/10.1145/159764.159763

[54] Judith S. Olson, Dakuo Wang, Gary M. Olson, and Jingwen Zhang. 2017. How People Write Together Now: Beginning the Investigation with Advanced Undergraduates in a Project Course. *ACM Trans. Comput.-Hum. Interact.* 24, 1, Article 4 (March 2017), 40 pages. https://doi.org/10.1145/3038919

[55] Cathy O'Neil and Rachel Schutt. 2013. *Doing Data Science: Straight Talk from the Frontline.* "O'Reilly Media, Inc.". Google-Books-ID: ycNKAQAAQBAJ.

[56] Steve Oney, Christopher Brooks, and Paul Resnick. 2018. Creating Guided Code Explanations with Chat.Codes. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 131 (Nov. 2018), 20 pages. https://doi.org/10.1145/3274400

[57] Samir Passi and Steven J. Jackson. 2018. Trust in Data Science: Collaboration, Translation, and Accountability in Corporate Data Science Projects. 2 (2018), 136:1–136:28. Issue CSCW. https://doi.org/10.1145/3274405

[58] Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53

[59] Jeffrey M. Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563 (2018), 145. https://doi.org/10.1038/d41586-018-07196-1

[60] Ilona R. Posner and Ron Baecker. 1992. How people write together (groupware). In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Vol. iv. 127–138 vol.4. https://doi.org/10.1109/HICSS.1992.183420

[61] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. 1–2. https://doi.org/10.1109/JCDL.2017.7991618

[62] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proc. ACM Hum.-Comput. Interact.* 2 (2018), 150:1–150:12. Issue CSCW. https://doi.org/10.1145/3274419

[63] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 32, 12 pages. https://doi.org/10.1145/3173574.3173606

[64] Adam Carl Rule. 2018. *Design and Use of Computational Notebooks.* Ph.D. Dissertation. University of California San Diege.

[65] Helen Sharp, Robert Biddle, Phil Gray, Lynn Miller, and Jeff Patton. 2006. Agile Development: Opportunity or Fad?. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems (CHI EA '06)*. ACM, New York, NY, USA, 32–35. https://doi.org/10.1145/1125451.1125461

[66] Aurélien Tabard, Wendy E. Mackay, and Evelyn Eastmond. 2008. From Individual to Collaborative: The Evolution of Prism, a Hybrid Laboratory Notebook. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work (CSCW '08)*. ACM, 569–578. https://doi.org/10.1145/1460563.1460653

[67] M. Rita Thissen, Jean M. Page, Madhavi C. Bharathi, and Toyia L. Austin. 2007. Communication Tools for Distributed Software Development Teams. In *Proceedings of the 2007 ACM SIGMIS CPR Conference on Computer Personnel Research: The Global Information Technology Workforce (SIGMIS CPR '07)*. ACM, New York, NY, USA, 28–35. https://doi.org/10.1145/1235000.1235007

[68] Darja Šmite, Nils Brede Moe, and Richard Torkar. 2008. Pitfalls in Remote Team Coordination: Lessons Learned from a Case Study. In *Proceedings of the 9th International Conference on Product-Focused Software Process Improvement (PROFES '08)*. Springer-Verlag, Berlin, Heidelberg, 345–359. https://doi.org/10.1007/978-3-540-69566-0_28

[69] Dakuo Wang, Haodan Tan, and Tun Lu. 2017. Why Users Do Not Want to Write Together When They Are Writing Together: Users' Rationales for Today's Collaborative Writing Practices. *Proc. ACM Hum.-Comput. Interact.* 1, CSCW, Article 107 (Dec. 2017), 18 pages. https://doi.org/10.1145/3134742

[70] Jeremy Warner and Philip J. Guo. 2017. CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1136–1141. https://doi.org/10.1145/3025453.3025876

[71] Judith D. Wilson, Nathan Hoskin, and John T. Nosek. 1993. The Benefits of Collaboration for Student Programmers. In *Proceedings of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '93)*. ACM, New York, NY, USA, 160–164. https://doi.org/10.1145/169070.169383

[72] Soobin Yim, Dakuo Wang, Judith Olson, Viet Vu, and Mark Warschauer. 2017. Synchronous Collaborative Writing in the Classroom: Undergraduates' Collaboration Practices and Their Impact on Writing Style, Quality, and Quantity. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2998181.2998356

[73] Xiong Zhang and Philip J. Guo. 2017. DS.Js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 691–702. https://doi.org/10.1145/3126594.3126663

Received April 2019; revised June 2019; accepted August 2019

Interactions for Untangling Messy History in a Computational Notebook

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

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

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

Keywords—exploratory programming, versioning, data science

I. INTRODUCTION

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

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

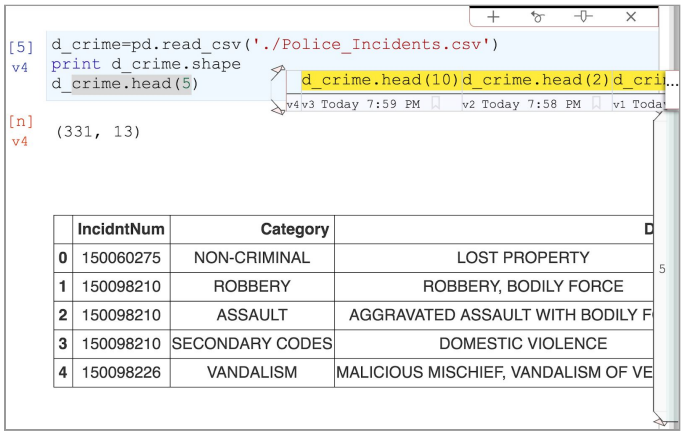

Figure 1. Verdant in-line history interactions. For the top code cell, a ribbon visualization shows the versions of the third line of code. In the output cell below, a margin indicator on the right shows that there are 5 versions of the output.

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

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

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

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

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

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

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

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

II. RELATED WORK

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

Provenance work: Provenance, tracking how a result was produced, has many different levels of granularity, all the way down to the operating system-level of the runtime environment [14]. In this work, we do not collect *absolute* provenance, since we only collect reasonably fine-grained runtime information about code, input, and output that is accessible from inside the computational notebook environment. The focus of our research is how to make provenance data *usable* to data scientists, and thus we focus on recording the history metadata most useful for data scientists at the cost of some precision. Pimentel et al. in 2015 created an extension to Jupyter notebooks that collects Abstract Syntax Tree (AST) information to record the execution order

and the function calls used to produce a result [13]. However, to retrieve this history, users must write SQL or Prolog queries into their notebook to retrieve either a list of metadata or a graph visualization of the resulting dependencies [13]. Instead of having users write more code to retrieve history, our focus in this work is to provide direct manipulation interactions which require far less skill from the user. Extensive prior provenance work has used graph visualizations to communicate provenance relationships to users [21], however graph visualizations are well known to be difficult for end-users to use [22], thus we avoid them.

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

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

III. VERDANT VERSION MODEL

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

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

A full version of the notebook is captured each time any cell is run. For efficiency, commits only create new nodes for whatever has changed, using reference pointers to all of the child nodes of the previous commit for whatever is the same. Versioning in this tree structure and at the AST level addresses many concerns of scale. For instance, imagine a data scientist Lucy has iterated on code for 257 different runs, but has only changed a certain parameter 3 times. Through AST versioning, Lucy does not have to sift through all 257 versions of her code with repetitive parameter values, but can instead simply retrieve the 3 unique versions of that parameter. Although AST versioning provides a great deal of flexibility to provide context-specific history, like Lucy's 3 versions of her parameter, it adds algorithmic challenges. Namely, each time Lucy runs her code, there is the full version A of the AST which is the last recorded version of the program and a new full version B of the AST that is the result of all of Lucy's new changes up to the point of the run. Matching two ASTs has been done previously, using heuristics like string-distance,

type and tree structure properties [26][27], however note that this matching has not been used in user-facing edit tracking before. Further, what is a correct match from a pure program structure perspective may not always match what is "correct" to the user. For instance, if Lucy changes a parameter `3` to `total("Main St.")`, Lucy may want to see the history of these two AST nodes matched, since both are serving the exact same role as her parameter, however since these are far in both type and string-distance, a traditional matching algorithm would *not* match the two. Refining this matching algorithm to match user expectation is an area for future work, thus for the purposes of the immediate design exploration, Verdant implements a simple Levenshtein string matching algorithm: if the token edit distance between two AST nodes is less than or equal to 30% of the length of the nodes, Verdant considers them a match.

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

V. INTERACTIONS FOR VERSION FORAGING

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

A. Ambient Indicators

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

B. Navigating Versions

The "ribbon display" shown in Figure 1D is the default way Verdant shows all versions for an artifact, lined up side by side to the right of the original artifact. Unlike existing code interactions like a linter or autocomplete, where a pop-up may appear in the active text to supply short static information, versioning data is comprised of a long ordered list of information and must continuously update as the data scientist runs their code. So in the ribbon visualization, because code

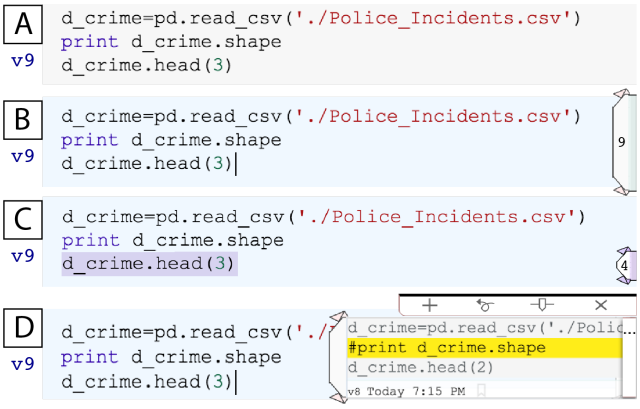

Figure 2. (A) no version information shown, (B) on selecting a cell an margin indicator displays how many versions of that artifact there are, (C) on selecting specific code the indicator updates to show how many versions of that specific snippet there are, (D) upon clicking the indicator, a ribbon visualization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

and cells in the notebook are read from top to bottom, the version property of an artifact is visualized left to right, with the leftmost version, which is shown in blue (Figure 2D), always being the *active version*. Here "active version" will always refer to the version of the artifact that is in the notebook interface itself and that is run when the user hits the run button in the notebook. Since the ribbon is a horizontal display, it can be navigated by horizontal scroll, the right and left arrow keys, or by clicking the ellipsis bar at the far right of the ribbon which will open a drop-down menu of all versions (Figure 2D). For navigating versions, note that Variolite made a different design decision, and had users switch between versions via tabs. However, as suggested by Variolite's usability study [4], as well as recent work on tab interfaces in general [30], tab interfaces do not scale well as the number of versions increases beyond a handful. On the other hand, choosing from a list display is not the most speedy for

retrieving an often used version if it is far down on the list. The ribbon display always shows the most *recent* versions first, making recent work fast to retrieve on the intuition that recent work is more likely to be relevant to the user's current task. For non-recent items, bookmarking is a standard interaction for fast retrieval of often-used items. Since robust history tools currently do not exist, we lack grounded data on which history versions a data scientists is likely to use. So to probe this question with real data scientists, we added an inactive bookmark icon to all versions, indicating that the user *can* bookmark it. We use this during our initial usability test to probe potential users on bookmarking, their use cases (if any) for it, and on various ways it could be displayed (see below).

C. Comparing Versions

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

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

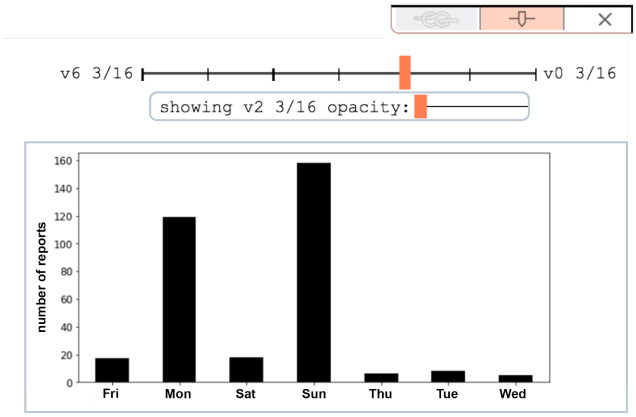

Figure 3. Timeline view. By dragging the top orange bar side to side the user can change the version shown. By dragging the lower orange bar, the user can set the opacity of the historical version they are viewing, in order to see it overlayed on top of the currently active output version.

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

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

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

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

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

D. Searching & Navigating a Notebook's Full Past

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

The list pane can be opened by the user with a button, and is tightly coupled with the other visualizations such that if the user selects an artifact in the notebook, the pane will update to list all versions of that artifact, and stay consistent with the current selected version. If no artifact is selected, the list shows all versions of the notebook itself. With a view of the entire notebook's history, the user can see a chronologically ordered change list beginning with the most recent changes

across all cells in the notebook. Say Lucy wants to retrieve a result she produced last Wednesday that has since been deleted from her current notebook. Either by scrolling down the list or by using the search bar to filter the list by date, she can navigate to versions of her notebook from last Wednesday to try to pull out the relevant artifact when it last existed. Alternatively, she can use the search bar to look for the result by name. Note that Lucy does not need to actually find the *exact* version she is looking for from this list. Using foraging, if she can find the old cell in the list that she thinks at some point produced the result she is thinking about, she can select that cell in the list to pull up all of its versions of code and output. From there, she can narrow her view further to only show the output produced on Wednesday. This method of searching relies on following clues across dates and dependency links among artifact versions, rather than requiring the data scientist to recall precise information that would be needed for a query in a language like Prolog [13].

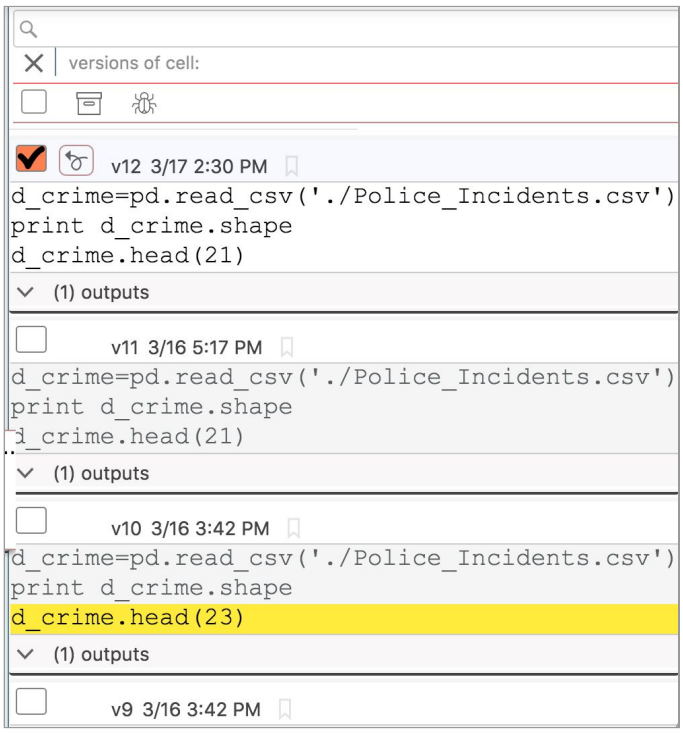

Figure 4. In the list view, the user can select one or more versions to act upon. With the search bar, the user can filter versions using keywords or dates.

VI. REMIX, REUSE, & REPRODUCE

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

A. Replay older versions

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

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

B. Output Recipes

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

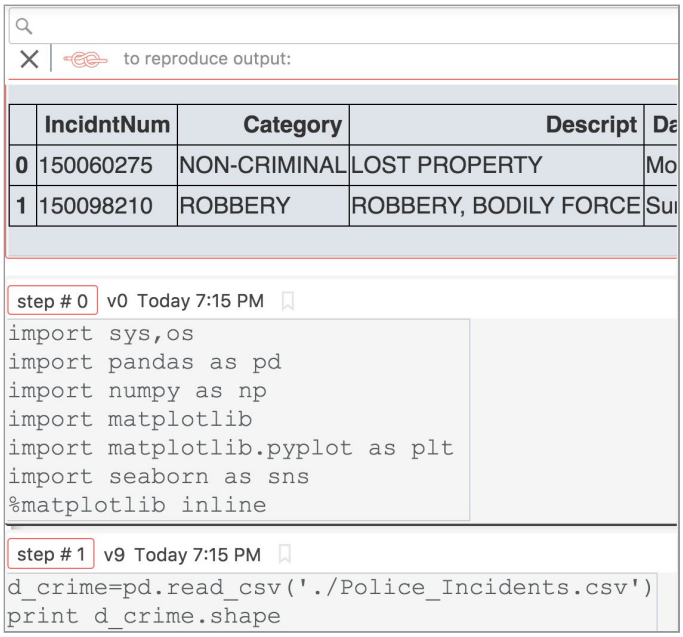

Figure 5. In the recipe view, a user sees the output they selected first, and then an ordered series of code cells that need to be run to recreate that output.

B. Trust and Relevancy of versions at scale

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

VII. INITIAL USABILITY TEST

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

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

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

All participants were able to successfully complete the tasks, suggesting at least a basic level of usability. Among even a small sample, we were surprised by the diverse use cases participants expressed that they had for the tool. P1 and P5 expressed that they would like to use the ribbon visualization of their versions about every 1-2 days to reflect on their experiment's progress or backtrack to a prior version. P2 was largely uninterested in viewing version history, but instead was enthusiastic about using the ribbon visualization to switch between 2 to 4 different variants of an idea. P3 was less interested in viewing version history of code cells, but greatly

valued the ability to view and compare the version history of output cells. P3 commonly ran models that took a long time to compute (so they only wanted to run a certain version once), and currently to compare visual outputs, had to scroll up and down their notebook. However, P3 did appreciate the ability to version a code cell, as a safe way of keeping their former work in case they wanted to backtrack later. Finally P4 primarily used notebooks in their classwork, and were very enthusiastic about using code artifact history to debug, revert to prior versions, and to communicate to a teaching assistant what methods they had attempted so far when they went to ask for help. P2, P3, and P4 expressed they would like to use the inline history visualizations "all the time" when doing a specific kind of task they were interested in, whether that be comparing outputs or code.

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

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

VIII. FUTURE WORK & CONCLUSION

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

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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

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

Systems, 2018, p. 174.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CHI 2019, May 4–9, 2019, Glasgow, Scotland UK

ACM ISBN 978-1-4503-5970-2/19/05…$15.00
https://doi.org/10.1145/3290605.3300500

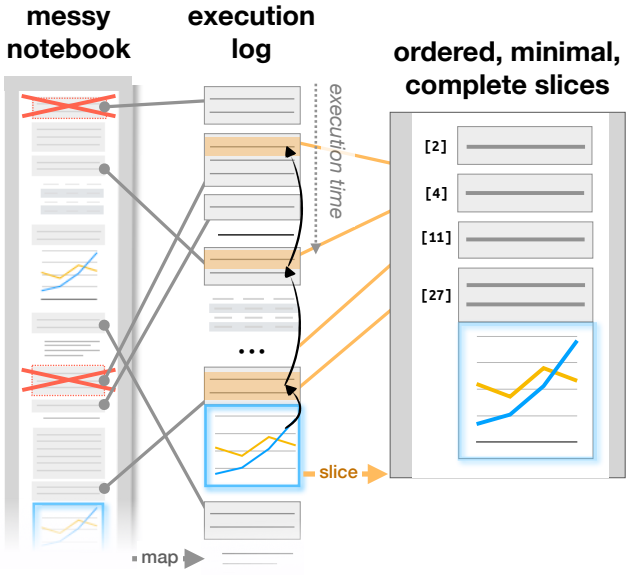

Figure 1: Code gathering tools help analysts manage programming messes in computational notebooks. The tools map selected results (e.g., outputs, charts, tables) in a notebook to the ordered, minimal subsets or "slices" of code that produced them. With these slices, the tools help analysts clean their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

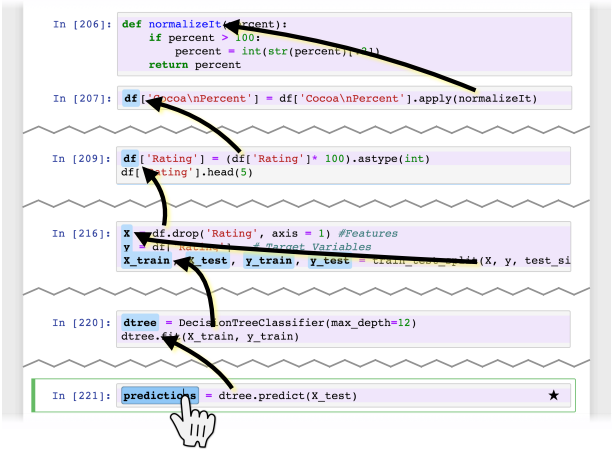

Figure 2: Finding relevant code with code gathering tools. With code gathering tools, an analyst can click on any result, and the notebook highlights in light purple just those lines that were used to compute the result or variable. The highlights appear throughout the notebook (which is condensed in 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

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

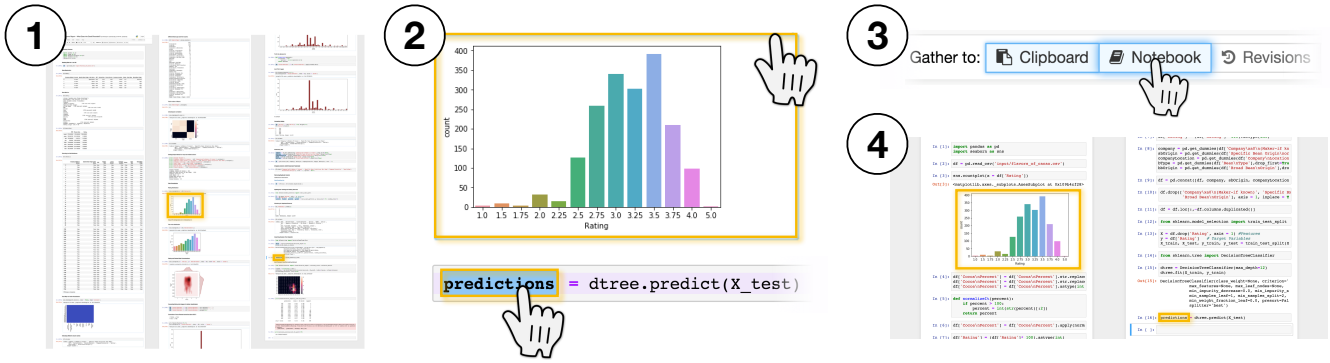

Figure 3: Cleaning a notebook with code gathering tools. Over the course of a long analysis, a notebook will become cluttered and inconsistent (1). With code gathering tools, an analyst can select results (e.g., charts, tables, variable definitions, and any other code output) (2) and click "Gather to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

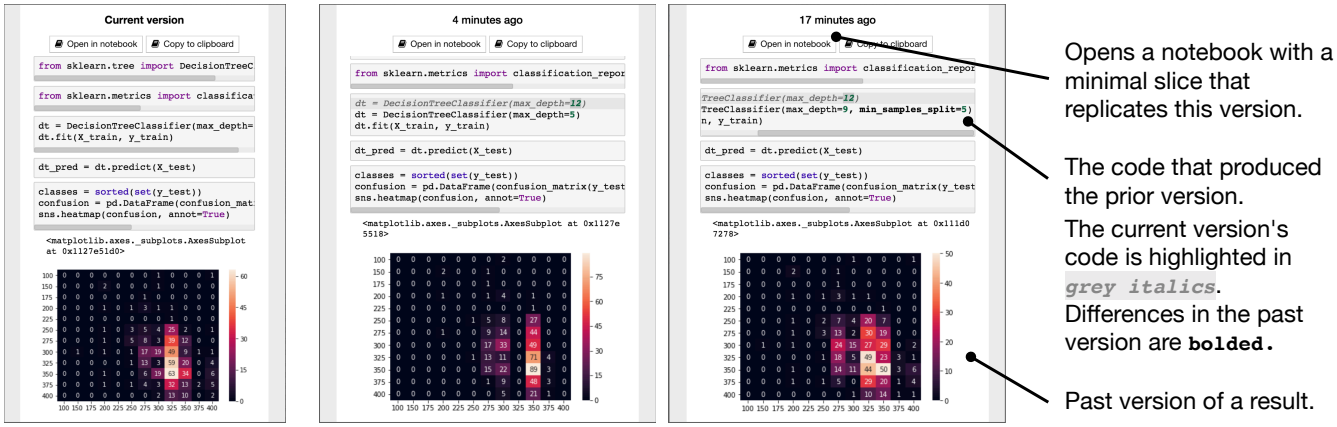

Figure 4: Comparing versions of a result with code gathering tools. When an analysts executes a cell multiple times, code gathering tools archive each version of the cell. When the analyst chooses the cell's output—say, the confusion matrix shown above—and clicks "Gather to Revisions," a version browser 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

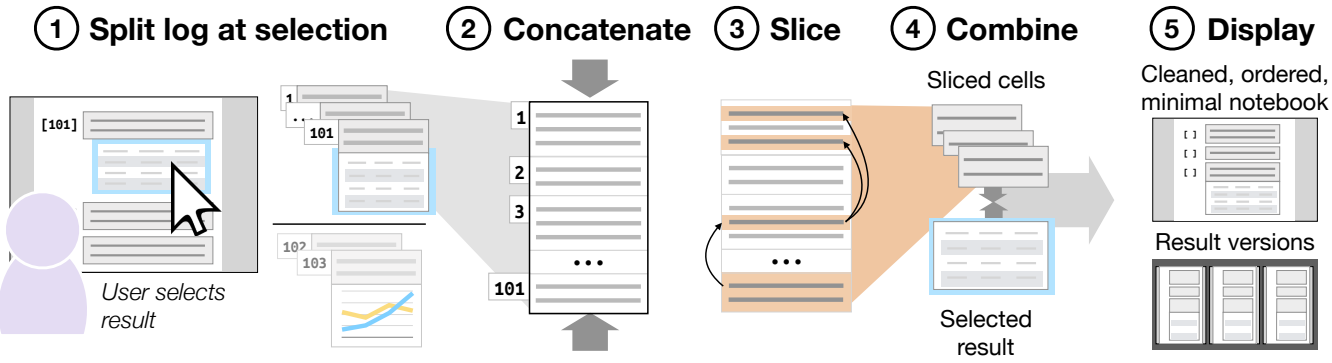

Figure 5: Implementation of code gathering. When an analyst wants to gather the code that produced a result, the code gathering backend splits the log of executed cells at the last cell where the analyst clicked a result, and discards the other cells (1), concatenates the text from the remaining cells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

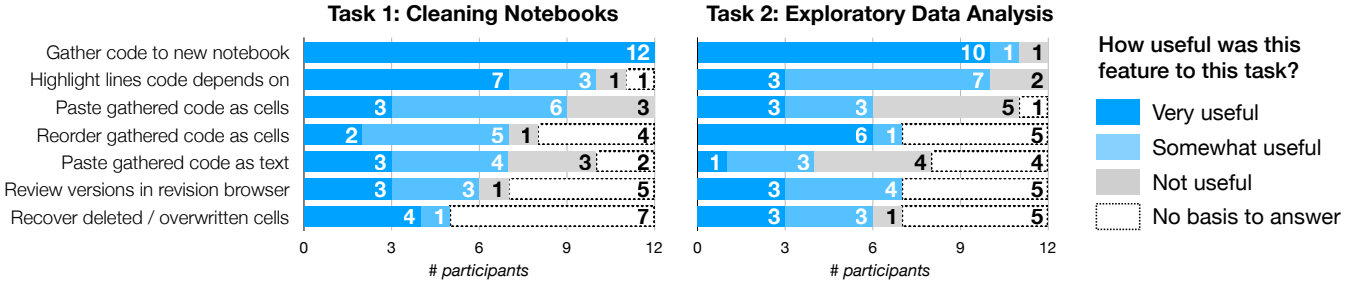

Figure 6: Analysts found code gathering tools most useful for gathering code to new notebooks, when they cleaned notebooks, and when they performed exploratory analysis. Analysts also appreciated dependency highlights, especially when they were cleaning code.

(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.