

Designing Effective History Support for Exploratory Programming Data Work

Mary Beth Kery
May 2021

Human–Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
mkery@cs.cmu.edu

Thesis Committee:
Brad A. Myers (Advisor), HCII, CMU
Bonnie E. John, Bloomberg L.P.
Nikolas Martelaro, HCII, CMU
Dominik Moritz, HCII, CMU, Apple Inc.

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy*

The research reported here was supported, in whole or in part, by Bloomberg L.P. and by the National Science Foundation (NSF) under Grants IIS-1314356, IIS-1644604 and IIS-1856641 to Carnegie Mellon University. All views expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

Copyright © 2021 Mary Beth Kery. All rights reserved.

Abstract

Why did you model the data that way? How do we reproduce this plot? Programming for data science or modeling is a highly valued skill today. Yet when data workers *experiment* with data by coding — an intensely iterative process called *exploratory programming* — the details of what they try along the way to a solution tend to get lost. Since experimentation underlies essential workflows in data analysis, machine learning, AI, and visualization this is a serious flaw. Ask any data worker today, and regardless of organization or years of experience, they have faced at least some results that cannot be readily reproduced, or mysterious data decisions missing a rationale. Modern best practices for managing experimentation take high human effort and still leave considerable room for error. With rising demand for responsibility and accountability of analyses and models, it is vital that people have proper support for documenting and answering *why* things were built the way they were.

This dissertation explores *history tooling* to support exploratory programming data work. To do this, we first conducted interviews, surveys, and design exercises with practitioners to learn about their needs and current workflows for experimenting today. We contribute two studies: **1)** a study detailing the mix of tools and ad-hoc methods data workers use to manage their experiments, and **2)** an investigation of how data workers use computational notebooks for iteration. Our results point to two key barriers: the manual effort needed to collect experiment history today is unsustainable, and recovering *semantic process information* out of a pile of history logs is far too cumbersome for practitioners to fit into their workflows today. We aim to help practitioners record their experimentation without any manual effort, and moreover, quickly recover history facts to answer rationale questions about their work.

Next in this dissertation, we design, build, and test new interactive tools to meet these design goals, over a 5 year iterative human-centered design process. We contribute: **1)** a series of 5 experiment history tool prototypes and 4 usability studies with practitioners, each of which illuminates a different aspect of the design space, **2)** a set of novel visualization and interaction techniques for concisely summarizing history, **3)** a fully implemented experiment history tool called Verdant, deployed in the wild as a computational notebook extension, and **4)** an observational study where data workers use Verdant during exploratory programming and afterwards to answer rationale questions about the history of their experiments. With Verdant, participants were able to answer 98% of history questions about their work in 1 minutes 26 seconds on average. All participants reported ways in which Verdant's style of history support would help in their own real life work practices. In the conclusions of this thesis we discuss the broader design space of experiment support tooling that rich history data enables.

Table of Contents

Abstract	1
Acknowledgments	6
Chapter 1: Introduction	9

Part I: Looking Close at How Data Scientists Use Programming to Experiment with Data

Chapter 2: Exploratory Programming as a Phenomenon	21
Chapter 3. Informal Versioning Behaviors in Data-centric Exploratory Programming	29
Chapter 4. Usage of Notebooks in Data-centric Exploratory Programming	38

Part II: Designing Better Support for Experiment History

Chapter 5: Related Work on History Systems	57
Chapter 6: Design Process Overview	69
Chapter 7: Interactive Snippet Versioning <i>Featuring Variolite</i>	73
Chapter 8: Capturing the Full Picture of an Experiment <i>Featuring Rose Quartz</i>	85
Chapter 9: Collecting & Modeling History in Notebooks	91
Chapter 10: Interactive Versioning within a Notebook <i>Featuring Verdant-1</i>	103
Chapter 11: Recovering Experiment History Facts at Scale <i>Featuring Verdant-2</i>	119
Chapter 12: Design and Engineering Iterations for Robustness <i>Featuring Verdant-3 & Verdant-4</i>	140

Part III: Putting Experiment History into Practice

Chapter 13: Designing a Realistic Usage Study	157
Chapter 14: Results & Discussion	

Part IV: Future Work & Conclusions

Chapter 15: Future Work	197
Chapter 16: Conclusion	203

References207

Appendices

Appendix A: Exploratory Programming Study Materials	221
Appendix B: Notebook Usage Study & Query Design Exercise Materials	232
Appendix C: Variolite Usability Study Materials	236
Appendix D: Jupytercon Scavenger Hunt Study Materials	244
Appendix E: Classroom Deployment Pilot Materials	259
Appendix F: The Verdant Study Materials	265
Appendix G: Query Design Exercise Results	293

Table of Studies

Study Name	Participant Naming	Appears in Chapter
Exploratory Programming Study	P01, P02,... P10	Chapter 3
Notebook Usage Study	N01, N02, ... N21	Chapter 4
Query Design Exercise	SP01, SP02, ... SP45	Chapter 4
Variolite Usability Study	Not named	Chapter 7
Verdant-1 Usability Pilot	D01, D02, ...D05	Chapter 10
Jupytercon Scavenger Hunt Study	J01, J02, ...J16	Chapter 11
Deployment Pilot	Not named	Chapter 12
The Verdant Study	E01, E02,... E11	Chapter 13, Chapter 14

Table of Systems

System name	Backronym	Appears in Chapter
Variolite	Variations Augment Real Iterative Outcomes Letting Information Transcend Exploration	Chapter 7
Rose Quartz	Recollection Of Serial Experiments: Quiet Utility Always Recording Tree-structure Zen	Chapter 8
Verdant-1	Versions Effortlessly Recorded, Displayed Around Notebook Tastefully	Chapter 10
Verdant-2	Version Explosion Rectified Diligently Anticipating Notebook Timewarp	Chapter 11

Verdant-3 	Very Exciting Re-Design Aiding Navigating Timewarp	Chapter 12
Verdant -4 	Versioning Experiments Reinforce Data Analysis Notebooks Temporally	Chapter 12

Acknowledgements

A PhD dissertation always has just a single author, as if all of this research journey was undertaken solo, but of course that isn't really true.

If custom allowed us to add a second author (without the validity of this doctorate being questioned, *oh no!*), that would be my excellent advisor **Brad A. Myers**. This document would not even exist if Brad hadn't convinced me back in 2015 to join the fantastic world of HCI for programming. I entered the PhD program as an enthusiastic 1st year thoroughly dedicated to researching *something* in HCI (possibly *everything* in HCI if I could). Brad's vision for the future of programming was so compelling that I became convinced that this is a research space in which I *can do everything* in a combination of HCI approaches from design, behavioral, technical and engineering. I signed on to Brad's lab 6 years ago and am ending my tenure all the more optimistic that HCI for programming has an important role in the future of programming. Along the way, Brad has been a great advisor in supporting the development of my work and professional development with a thoughtful attention to detail. Up to the very last word of this dissertation, the writing would certainly be a whole lot poorer if not for Brad's meticulous attention as editor.

Naturally, the true third author of this work is the inimitable **Bonnie E. John**. While Bonnie could not officially be a PhD advisor, since she is busy powering UX design over at Bloomberg L.P., it was sometimes a very near thing. A large portion of work in this thesis was done in collaboration with Bonnie, and is much better for it. If I began qualitative and UX research with all that unsteady beginner's enthusiasm with which I started HCI research, I have Bonnie to thank for relentlessly shaping my efforts towards methodological rigour. Bonnie's mentorship has been invaluable, from connecting me with research study opportunities to providing the funding, space, and design support at Bloomberg to develop Verdant. Finally, Bonnie has been a powerful role model for my own life: showing that you can have a pack of dogs, a deck of interesting hobbies, and some babies, all while balancing an impactful research career.

Last but not least, the dual fourth authors of this dissertation would be my final PhD committee members **Nikolas Martelaro** and **Dominik Moritz**. Nik and Dominik joined this research work at the thesis proposal phase, offering fresh perspectives on the work. Dominik brought a strong domain understanding of data work tools and his own experience developing open source programming tools to advise in the final stages of my tool Verdant. When Covid-19 hit, Nik was absolutely instrumental in not only offering his experience with running remote technical studies, but also single handedly providing full server setup and support for the Verdant Study. I cannot thank Nik enough for helping get that final study off the ground when I was really stuck with it. Both Nik and Dominik, with all their experience with systems HCI research as junior faculty, have been extremely helpful in career development advice.

Then, there is everyone else. Everyone who didn't write a word, and who may never read this dissertation, yet made it happen all the same.

First, there are my everyday companions, who have weathered this PhD journey with me. I would like to thank my dogs, **Ollie**, **Luna**, and **Lo-fi**, along with my cat **Taro** who always made

working from home fun. My insanely impressive and hardworking spouse **Ken Holstein** has been my coworking buddy in coffee shops and airports and deck chairs all over the world any hour of the day or night from day 1 of the PhD. There is no succinct “thank you” to cover my love and gratitude. It’s been a joy to support each other and build a family together in our own pocket of this vast and unforgiving cosmos. Our daughter **Amelie Kery-Holstein** I have to thank for lending us fresh joy and optimism for life during this past final year of the dissertation. How could I stay gloomy about minor research pitfalls when Amelie, my flesh-and-blood thesis baby born amidst a deadly global pandemic, quarantine, and an election year, was the happiest person in any room? Thank you Amelie. You didn’t have to be an easy going baby, but you were and it helped. Thank you as well to Amelie’s own outrageously good childcare support crew **Adora Holstein, Stephen Holstein, Alisa Williams, and Wylder Williams**.

This past year of pandemic, my community at CMU has often felt a hundred miles away and a long time ago. In the before-times, my friends and colleagues at the Human-Computer Interaction Institute were a foundation of support bolstering my first 4.5 years of the PhD. I grieve what felt like losing that community suddenly and permanently in the final 1.5 years taken by COVID. Still, I hold immense gratitude for everyone I got to know in my PhD years: my office-mate besties **Franceska Xhakaj** and **Fannie Liu**, as well as my entire PhD cohort including **Felicia Ng, Kristen Williams, Siyan Zhao, Judith Uchidiuno, Micahel Madaio, Joselyn McDonald, Julian Ramos, Joseph Seering, and Michael Rivera**. Thank you to my PhD seniors **Anna Kasunic, Sauvik Das, Dan Tasse, Qian Yang, Gierad Laput, Rebecca Gulotta, David Gerritsen, Jenny Olsen, and Judeth Oden Choi** for making the HCII wonderful and so welcoming too. Also thank you to my PhD juniors **Amber Horvath, Samantha Reig, Yasmine Kotturi, Lea Albaugh, Erica Cruz, Michal Luria**, and pretty much every current HCII PhD student for being compassionate and awesome. A special thanks to **Queenie Kravitz** and **Je. Bigham** who keep the PhD program running and have gracefully put up with my occasionally aggressive advocacy for improving PhD work conditions through the years. As the HCII community rebuilds after COVID, I have every faith it will be just as wonderful as ever because of all the people there.

A big thank you to everyone who paved the roads and opened up opportunities to lead me to a PhD in the first place. Those are my undergraduate research mentors at Wellesley, UC Berkeley, and NC State, as well as my high school programming teacher. My earliest mentors taught me the value of being generous with your time and generous with giving proper credit to lift up the careers of young computer scientists –and giving all that effort even when it doesn’t directly benefit your own career. That generosity that I received, I hope always to reciprocate in my own mentoring.

Finally, thank you to the rest of my family (both biological and chosen family) who value education so much: my parents **Jo-Ann** and **Sean Kery**, my siblings **Caroline Kery, Rachel Kery, Anna Kery, and Ren Smith**. My grandmother **Jane Ward** was the very first person to encourage me to go for a PhD, though she passed away the summer before I started. If she were still here, I would need to thank her for being one of those few people to read this thesis cover to cover, as I know she would.

Chapter 1: Introduction

With the wide availability of cheap sensing, metrics, and computational power today, data is being collected continuously in nearly every facet of human society. With a tremendous amount of data, people and organizations want the ability to *use it* in ways that generate value. Making use of data today requires a lot of human effort [Terrizzano et al. 2015, Sambasivan et al. 2021]. From data collection to data cleaning, exploration to modeling, amidst the myriad of tools, algorithms, and new techniques advancing all the time, *there are humans at the center of every data e. ort*, applying their expertise and human judgement to create analyses and models that work responsibly and well [Mittelstadt 2019, Arrieta et al. 2020]. Back in 2012, the Harvard Business Review famously called the new burgeoning job title “data scientist” the “Sexiest Job of the 21st Century” [Davenport & Patil 2012]. In reality, people across science, math, and almost any domain had been computing with data since the very dawn of computers. Back in the 1940s programmers were calculating ballistics analytics for the U.S. Army¹. Now, the rise of high-paying jobs like “data scientist” or “machine learning engineer” in the 2010s signified a new demand and recognition of just how important skilled data work had become. Then in 2020 Forbes published “AutoML 2.0: Is The Data Scientist Obsolete?” [Fujimaki 2020] and alarmists everywhere were projecting that society had advanced so far in machine learning (ML) that now AutoML techniques could fully replace human data workers with automation. However, a flurry of thought pieces and research articles [Wang et al. 2019, Xin et al. 2021] were quick to demonstrate that this hype was not realistic. AutoML was another helpful tool in a data worker’s toolbox, and a stop-gap for organizations that didn’t have enough human data scientists, but even though AutoML can optimize over millions of possibilities, the guidance of *human intelligence* was still needed [Wang et al. 2019, Xin et al. 2021]. Why is that?

In this research we focus on the human act of **exploration and experimentation** with data, machine learning, AI, visualization, and everything under the umbrella of data work. Exploration and experimentation are a vitally human effort, we argue, because no matter how much automation we provide to assist in exploring a broad decision space, human sensibilities such as domain knowledge, common-sense reasoning, social reasoning, and sensitivity to context are required to guide which directions are the most promising. Rather than any one specific task in any data or ML pipeline, experimentation is a cross-cutting attribute of many, many, jobs. For instance, in Microsoft’s diagram of a data science team process (Figure 1.1), nearly every step from feature engineering [Zheng & Casari 2018] to data cleaning [Kandel et al. 2011] requires experimentation to figure out what works and what doesn’t. Ultimately “what works” is up to human discretion too, a goal post negotiated through the process of experimentation.

¹ <https://www.computerhistory.org/revolution/birth-of-the-computer/4/78>

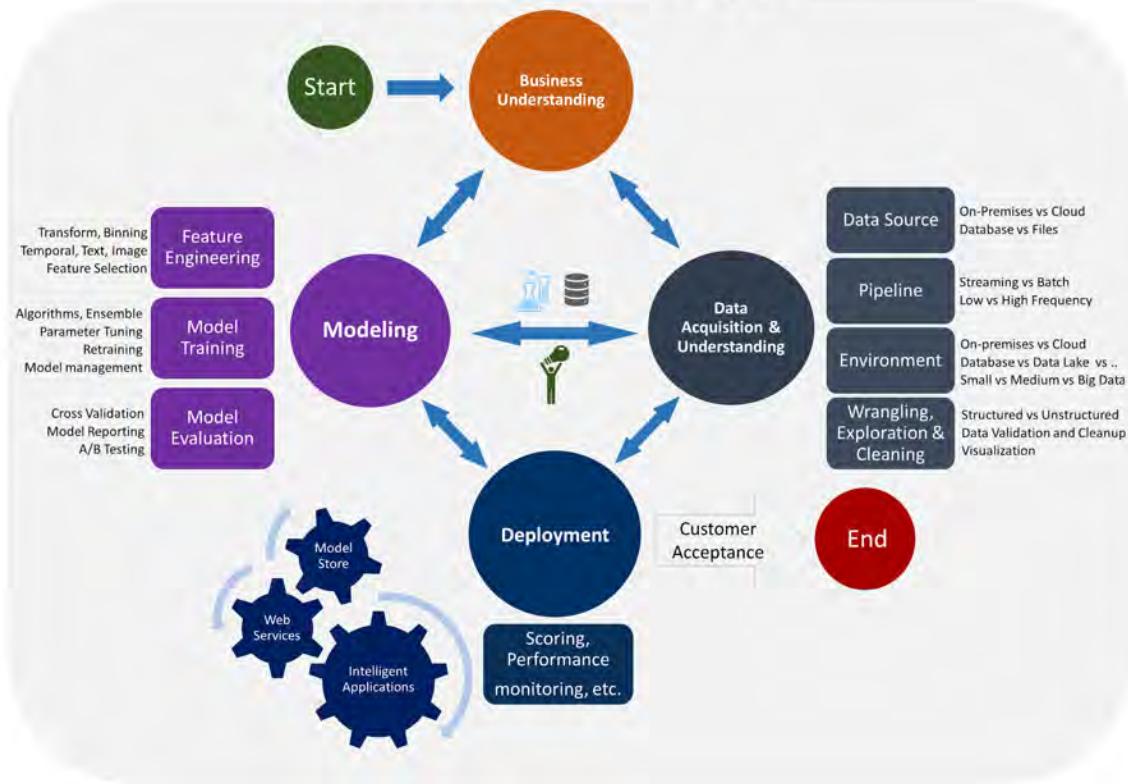


Figure 1.1. A diagram of the data science process for a team from Microsoft [Microsoft 2020].

If data science is the “*Sexiest Job of the 21st Century*”, our research shows that experimenting with data through code is decidedly *not sexy*. It’s often messy, hacky, and an assortment of code organization practices and haphazard schemes to keep track of plots and model metrics and analyses in some homegrown way so that someone *might* be able to find them later. “No,” you might object: “certainly *good* data scientists are *professionals* and do their work *well*.” My first research study with data scientists was an interview and survey study (Chapter 3) where, out of personal and professional curiosity, I wanted to learn about how data scientists conduct and manage their code explorations, at the very literal level of looking at their code, files, data, and notes. It became apparent that no one has a satisfying solution when it comes to managing code experimentation, no matter the experience of the practitioner or which organization they come from. To illustrate, here is a portrait of a *good* data exploration workflow, based on the dozens of data scientists we have spoken to in the course of this research from 2015–2021:

Kyoko is developing a machine learning model to predict whether a student is likely to drop out of University W. She has a bunch of different Python scripts for different data analyses of student data, as well as a script for her model. Kyoko is a skilled programmer, and backs up her code on version control with Git. In addition, she also has a folder of data versions: `student_data_raw.csv`, `student_data_cleaned.csv`, `student_data_norm.csv`, which she keeps on DropBox because the data is too large to version with Git.

Similarly, Kyoko has a folder full of interesting plots she's generated at various points of her analysis, which she also keeps on DropBox. Some of the plots were generated a long time ago, with older versions of the code.

Locally on her machine, Kyoko is currently busy experimenting with her model, so she has multiple copies of her modeling script: `model_v1.py`, `model_v2.py`, `model_v3.py`, `model_trees.py`, `model_biases.py`, `model_sensitiveFeat.py`. Kyoko is the only person who knows what those file names mean and it's not obvious what the changes between them are. To keep model comparison principled, her model script log model metrics to a csv file `results.csv`. This csv file is backed-up in version control with her code in Git. In addition, Kyoko keeps detailed notes in a Google Doc describing her experimentation and the different possibilities she's trying.

Now, Kyoko in this example is remarkably well-disciplined with managing experimentation and still it is easy for us to say: this is not great. Project information is scattered across multiple locations and it will take some coordination to figure out which data versions and which script versions generated which plot versions. But, is it good enough? The very real consequences of *not* keeping track of experimentation are lack of reproducibility and lack of explanation: plots that can't be reproduced, modeling rationale that no one remembers, data cleaning choices that no one can really justify 6 months later. Even with a relatively disciplined process like Kyoko's, it's never a question of *all* of her experimentation reproducing —some of her plots will almost certainly mysteriously not have code that creates them because that specific version wasn't committed to Git. Data work includes so many fast paced changes that many iterations are left unsaved by conventional tools like Git. The available best-practices leave plenty of room for error. For instance, the fabulously pragmatic and oft-cited “*Good enough practices in scientific computing*” [Wilson et al. 2017] suggests:

- Back up (almost) everything created by a human being as soon as it is created
- Keep changes small
- Share changes frequently
- Create, maintain, and use a checklist for saving and sharing changes to the project
- Store each project in a folder that is mirrored off the researcher's working machine

Figure 1.2 “Keeping track of changes” from Good Enough Practices in Scientific Computing [Wilson et al. 2017]

To follow Wilson et al.'s guidelines is *a lot of extra effort* that, importantly, slows a person's actual code iteration down (to stop and record and take notes) without helping them get any closer to their goal (see the Exploratory Programming Study, Chapter 3). The tricky thing about history for data exploration is that no person has the perfect foresight to predict what they will later need to return to, reproduce, or explain. A data scientist could add an hour to their work time carefully documenting every feature engineering variation they try, only to throw out the whole model the next day and never touch that history again (see the Exploratory Programming

Study, Chapter 3). There's inherent risk and cost to following good history practices in exploration. To prioritize *doing the actual exploration work*, we have found in multiple studies (Chapter 3 & 4) and confirmed by prior work [Guo 2012, Rule et al. 2018], that most practitioners are making rational compromises: get more work done by cutting corners on recording history, and then accept the cost that from time-to-time they will need to completely re-code something from scratch to reproduce it. Re-coding someone isn't always bad. It's possible that with re-coding a prior analysis, a practitioner may reflect on it or improve code quality the second time around. But re-coding is not an ideal safety-net, because it requires that a practitioner *actually remembers* how they got a result in the first place.

We can do better. It is surprising that after decades of data work, such a low-level workflow issue as managing experiments has such lackluster support. There has been no shortage of articles [Zhang et al. 2020, Xin et al. 2018], workflows and tools [Amershi & Conati 2009, Patel 2010, Guo & Seltzer 2012, Pimentel et al. 2019, Kunal et al. 2019, Wang et al. 2020] proposed for this issue, so why does the problem remain? We believe that what has been missing is a tight pairing between technology design and *behavioral* understanding: it's hard to design effective history support when as a community we don't really know how people do exploratory programming for data work in the first place! Leveraging close behavioral study of real work practices, we propose that an effective solution can be found by redesigning software version control, into new history algorithms and interactions tailored for exploration and experimentation. This brings us to the thesis of our current research:

Thesis Statement

Data work frequently involves exploratory programming which requires a new kind of versioning for history and new interaction techniques for exploring that history, which can help data workers more effectively answer their questions about what they explored.

To inform history tooling design, in this dissertation we investigate real practitioners' current needs, beliefs, and workflows around experimentation in a series of studies:

- **Exploratory Programming Study (Chapter 3): *How do data workers do exploratory programming today?*** In this study we interview and survey data practitioners about their beliefs and practices around experimentation. We center our investigation around the *artifacts* like code, data, files, and output that data workers create in their work, and how they manage these over time. We find that exploration adds risk of investing in an idea that may fail or be discarded. For this reason many practitioners prioritize “finding a solution over writing high-quality code”. Practitioners commonly use “informal versioning” practices like commenting or copying code to keep alternatives during experimentation.

- **Notebook Usage Study (Chapter 4):** *Do computational notebooks change how people experiment?* We conducted one of the first studies investigating how real data workers use computational notebook coding environments to experiment. We find that notebooks help practitioners author a story of their experimentation, but ultimately leave practitioners wanting more substantial history support. Notebooks are incompatible with conventional history tools like Git, requiring special work-arounds. Practitioners who try to narrate a record of *everything* they try into their notebooks end up with a complex and unreadable document. Other practitioners turn to “informal versioning” practices like duplicating their notebooks and cells to keep history.
- **Query Design Exercise (Chapter 4):** *What kinds of information would data workers seek out of their experiment history if they had it?* To answer this, we have practitioners ideate questions that they would want to ask a magical history oracle about their own work. In this design exercise, we collected 125 history questions generated by data scientists. This collection of history questions provides us a benchmark for how successful our design of history support is: we hypothesize that genuinely good experiment history support should allow data scientists to answer most if not all of these 125 questions.

Engaging closely with data scientists and their real practices allows us to find what data scientists would want out of actually good experiment history, summarized below in Table 1.1.

Table 1.1 An overview of behavioral studies and the design requirements for history systems we found in each.

Study	Design Findings
Exploratory Programming Study Chapter 3	<ul style="list-style-type: none"> + Content smaller than an entire code file should be considered for versioning. These smaller snippets contain important content like plots, models, and individual analyses. + Participants' copying and commenting behavior suggests that it is valuable for practitioners to have easy access to multiple versions of the same thing in their workspace. + Experimentation involves code and non-code artifacts, typically stored in separate places. Help users see how artifacts relate over time. + Experimentation is often too fast paced for practitioners to reasonably record. Automatically record what a user needs to replicate their experiment.

<p>Notebook Usage Study Chapter 4</p>	<ul style="list-style-type: none"> + History can provide a safety net for organizing the notebook. Since practitioners often curate as they work, removing less successful data work, a clear value proposition for history support is that we will be able to <i>preserve</i> these discarded parts of work while letting users keep a tidy notebook that contains just their most recent work. - “Expand then reduce” cell iteration means that cell-level versioning will be insufficient. Since practitioners develop code across multiple cells before combining it into a single cell, a code chunk may have provenance in multiple cells. + Most notebooks have a narrative cell structure top to bottom. A historical cell’s <i>location</i> in the notebook is likely to carry some semantic information about what the cell is about. - Explanatory notes are minimal during exploration. Many practitioners do not take the time to add markdown notes or comments at the stage of active development
<p>Query Design Study Chapter 4</p>	<ul style="list-style-type: none"> + Practitioners recall different kinds of textual, visual, date, location, output related information that is important to what they want to know from history. Help users find history facts using different kinds of remembered cues

These behavioral study contributions only get us to the point of design hypotheses: what we think actually good experiment history would look like. To test our hypotheses we also actively create our history support ideas such that we can test them out with real practitioners.

Based on these findings, we design new technologies to A) provide effortless interactive version control for data workers experimenting in code, and B) provide quick ways for people to ask and answer questions about their past experiments. This design process involves:

- **Paper prototyping** to rapidly test out different tool layouts and visualizations of history.
- **3 functional prototype history tools** that each provide different ways that a data scientist might use their experiment history. These prototypes are engineered and implemented to the point that they collect and display actual history data and can be used for real programming tasks.

- **5 usability studies** to test the efficacy of our history tooling designs by having real data practitioners use our prototypes for exploratory programming tasks.
- **A general history model** for collecting fine-grained experiment history data from a computational notebook.
- **1 deployed history tool, called Verdant**, for quickly retrieving experiment history facts in computational notebooks. Verdant is the culmination of several years of iteration and engineering and focuses on providing quick ways for data scientists to answer questions from their own experiment history.

Given the breadth and depth of design iterations we performed in this thesis, it is worth previewing some of our findings on what makes this design space challenging and what are the design barriers that make effective experiment management support so evasive in today's tooling. Design findings for each prototype tool are also summarized in Table 1.2.

The issue of *recording history* first appears quite amenable to automation. We might write a script that automatically saves and commits a data scientist's work to version control at regular intervals. Even for Kyoko's more complicated setup, we could write a script to save her plots to a folder, scripts to git, and so on to ensure all her results are always reproducible. Would this solve the problem? **No.** It turns out that recording history is just one requirement. After all, there are plenty of data scientists who do a respectable job of manually recording their own experiment history. Disappointingly, a pile of history is not necessarily usable or useful on its own. More often, experiment history is used today as emergency backup for the rare case that someone's computer breaks and left untouched otherwise.

A key complication is that data science experimentation quickly generates a large number of versions that can be too dense from which to draw information. For instance, an early test we did was to record a version each time we achieved a new result while coding a python script for a basic machine learning classification task. Within about 1 hour, the code had been edited and run 302 times! Lists of versions are highly susceptible to the long repetitive list problem [Ragavan et al. 2016]. Essentially, if there is a long list of similar variants of the same document, it is a laborious process for the user to search through them [Ragavan et al. 2016]. For pure code questions, a Git expert user may be able to use Git bisect or blame to track down where specific code changed. However for visual artifacts like plots or fuzzier questions like "*why did I discard this data feature*", the user is pushed into a tedious brute force search, reviewing version after version until they find the information they need. As a participant from our Notebook Usage Study (Chapter 4) put it: "*it's just a lot of stuff and stuff and stuff.*" If answering a quick historical question would take a disproportionately long time, a data scientist will not do it (Chapters 3&4).

Therefore, beyond recording the history, we need to provide users with *easy and effective tools to examine or ask questions about their history*. With this design goal, we first sketched some easy interactions a user might do to quickly check something from history, and just as quickly return to their actual experimentation. Although each systems chapter of this thesis (Chapters 6-12) starts with paper prototyping, **static prototyping is insufficient in situations involving complex dataflows**. Since the history data itself, the amount of it and its dense and repetitive

nature are a major part of the usability challenge for experiment management, we cannot evaluate the viability of our design ideas without seeing them in action with the history data. Thus Chapters 6-12 also include substantial engineering effort to create *functional prototypes* that are viable to the degree of evaluation needed. In Chapter 9 we cover the technical details of creating a history database of experimentation, and discuss a crucial point in this work, that data too is also designed and iterated on. Instead of first collecting experiment history from data workers *and then* designing user-interface support against a monolith history database, we work in the opposite direction. We first sketched out what ideal user-interface design we would like to support and what we would like data workers to be able to do in an imagined “actually good experiment history tool”, and then designed how data needed to be collected and stored to support those interactions in real time. Later in the design process, both data and the user interface were iterated together, based on changing needs in the user interface and performance constraints on the data side (Chapter 9).

Our early prototypes, Variolite and Verdant-1, are implemented just to the degree that history data could flow through the user interactions of each tool, so that we could test these interactions with real data workers. With our prototype Variolite (Chapter 7), we show that **data workers highly value the idea of interactive and in-situ version control**. With our prototype Verdant-1 (Chapter 10), we show that data workers value in-situ and interactive history that **gives context to experimentation by relating the history of code, outputs, and notes all together**. At this point we switched focus to tackling the issue of overly dense experiment history at scale. This took substantially more design and engineering time to create an experiment history tool suitable to function for days and weeks of experimentation work. The fruits of this effort was Verdant-2 (Chapter 11), and in the Jupytercon Scavenger Hunt Study (Chapter 11) we show that **our interactive history visualizations allow data workers to quickly answer history questions —even about someone else’s data work they are seeing for the first time**.

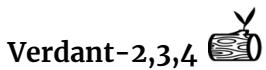
Next came further design iteration in Verdant-3 (Chapter 12) and a deployment pilot where we used Verdant-3 in a classroom setting. The finale of this design process is a deployed history tool Verdant-4 (Chapter 12). In order to evaluate to what extent Verdant-4 is in fact an “actually good experiment history tool” we engineered Verdant-4 to the point that it can be used by data scientists in the wild. Then, we conducted the Verdant Study (Chapter 13 & 14) where we have data scientists do exploratory programming with Verdant-4, and then later in a separate session, answer history questions using Verdant-4. Our results provide evidence that **with Verdant, data scientists can quickly answer history questions about their own analysis and model work with low effort**. In the final design, participants had a 98% success rate when using Verdant to answer history questions (Chapter 14).

So: do we believe we have achieved an actually good experiment history tool? As the author and lead researcher on this dissertation work I personally believe: yes. By today’s standards at least, we contribute a variety of interactive history tooling with highly promising empirical results, and contribute a detailed analysis of the problem space that carefully considers how real data scientists work. However, of course we also believe that *decent* history management for data work experimentation is really just the beginning of what experiment history could allow data workers to do. Beyond just being able to effectively answer questions about your own past data

work, in Future Work (Chapter 15) we discuss directions in experiment summarization, experiment recommendation, collaboration, and a variety of ways history may be used to make the human activity of experimenting with data more effective and more transparent.

Table 1.2 An overview of systems and design findings from each.

System	Design Findings
 Variolite Chapter 7	<ul style="list-style-type: none"> + Supports specific fine-grained versioning: allows creation and access of multiple alternative versions of individual snippets within a code editor. - Keeping versions in UI tabs does not scale well: while a tab interaction appeared easy-to-understand during prototyping, we found we could only display 3–5 named tabs in the available screen space, which is too few versions for realistic usage. + Supports continually shifting focus: by watching people work over time, we learned that regions of interest a user wants to version <i>continually shift over time</i> as the user tests out different hypotheses and develops different ideas. History needs to be <i>already available</i> for whatever content the user wants to look at. + Version all or nothing: in usability testing we found that versioning just <i>parts</i> of a code file leads to A) an overly-complex combinatorial code file and B) gaps in history where users later want history of code that was not versioned. We learned that the safest approach to avoid these issues is to just version everything. If we version all of a user's code file for each experiment, we can later retrieve the history of any specific portions of the file the user wishes, while simplifying the structure of the history we store.
 Rose Quartz Chapter 8	<ul style="list-style-type: none"> + Helps users see the relationship between code and non-code artifacts: An “experiment” is often a combination of changes made to code, data, or input parameters, with results in forms of output, data, or notes. + Helps users detect what changed from one experiment to the next: Experiment history is highly redundant because often the difference between one experiment to the next is only a small change. + Anticipate scale: experiment history is fast paced and quickly accumulates dozens of versions at the rate at which a data worker runs their code. Anticipate roughly 1

	<p>version per minute of work time.</p> <ul style="list-style-type: none"> - Capturing “an experiment” from different tooling sources is extremely difficult: Since data workers work in a variety of languages, tooling environments, and domain-specialized data analysis tools, instrumenting every possible tool that a data worker might incorporate into their analysis workflow is an enormous undertaking. There are also major privacy and security pitfalls. Alternatively, finding a central tool that already combines a workflow is much more tractable. For this reason, we focus on computational notebooks because they offer a rich workflow in a single tool.
 Chapter 10	<ul style="list-style-type: none"> + Inline access of the history of any artifact is wonderful: the ability to click on specific content and see its specific history is consistently popular with users of our prototypes. The appeal is that users are able to see the history of the content in front of them without having to search or skim through a bigger pile of history logs. + Help users reproduce results as a “recipe”: In our pilot usability study and reception by the research community, the metaphor of reproducing output as a “recipe” was easy for users to understand. - Inline history visualization makes it difficult to see how versions of different artifacts relate: we found in prototyping that showing all the versions of one artifact A and elsewhere showing all versions of a different artifact B allows users to examine A and B individually but tells the users very little about how the histories of A and B relate. Different UI approaches are needed to make history relationships clear. - Inline history visualization does not scale well: we found that inline history visualization works best for comparing just 2 versions of the same artifact side by side.
 Chapter 11,12, 13, 14	<ul style="list-style-type: none"> + An activity stream gives users confidence that their experiment history is being preserved: users in the summative Verdant Study found the activity “minimap” view to be a helpful companion as they programmed, because the visualization updated in real time to indicate what parts of their experiments had been recorded. This gave users confidence that the system was working.

-
- + **Give users the ability to switch between history detail and history context to understand relationships:** one of the most successful interactions for finding history information we observed in the Verdant Study was when users paired the Artifact Detail view with the Ghost Notebook view. Users browsed specific artifact history in the detail view, and then used a side-by-side Ghost Notebook to see how each version had occurred in historical context with other artifacts and output.
 - + **Structure history to minimize scrolling through lists:** in our Jupytercon study, users became frustrated (and often gave up!) if they believed that they needed to examine every version in a list to identify the right one.
 - + **Give the user multiple ways to find the same thing based on what they remember:** since Verdant has multiple ways of searching and visualizing history, there is no single “right” way to find something. We saw this benefit users in our study because regardless of which avenues a participant tried, they were able to reach the desired information.
 - + **Treat visual finding for visual artifact history differently:** In practice we found that plots often do not have descriptive keywords in the code that accompanies them, so they cannot be found with a textual keyword search. It is worth designing search features specifically for finding plots, images, and other visual-only artifacts.
-

Part I: Looking Closely at How Data Scientists Use Programming to Experiment with Data

INTRODUCTION TO PART I

In Part I of this dissertation, we first study the phenomenon of exploratory programming in Chapter 2, and then how exploratory programming plays out in real life by real practitioners in Chapters 3 & 4. From Chapter 2 we gain insights, primarily through a literature review, about the specific ways in which exploratory programming is different from normal software development. This is important because conventional history support for code, e.g. Git or SVN, is designed for software development. Exploratory programming departs from typical software development by prioritizing *fast iteration* of a *broad space of possibilities* instead of *careful engineering* upon *fixed goals*. This is reflected in how we design history support. For instance, a commit in Git typically represents a version in which a specific engineering goal is achieved, e.g., “fixed bug #456”. What would commits look like if instead versions represent specific ideas explored?

To answer questions about what history support *should look like*, we engage with what practitioners are dealing with today. In Chapter 3 we look at practitioners broadly engaging in exploratory programming for analysis or modeling, and look at how they use, abuse, or craft workarounds for history support available today. We find that conventional version control like Git is missing support for many of the small fast-paced explorations practitioners do that happen at units smaller than the whole file: for instance code snippets that represent models or plots. Whereas conventional version tools like Git preserve full snapshots of a user’s work, this isn’t the only viable approach to history. In Chapter 4 we look at computational notebooks, like Jupyter notebooks, to see if practitioners are able to author *narrative histories* of the experimentation they do. Beyond a conventional Python or R script file, we find that notebooks do help users structure a narrative of the experimentation process – but only to an extent. Notebook users typically curate their notebooks to show how they reach their *final result*, but for clarity of storytelling, users typically discard less successful experimentation they do along the way. Notebook users actually face *more* difficulty than conventional script users in that conventional versioning tools like Git currently perform poorly with notebooks. So although notebooks users gain storytelling as a history-keeping tool, they lose the benefit of a Git-model of history, and ultimately use many of the same workarounds as ordinary script users to make up for the missing history support they need. In both of these chapters, we detail how the specific workaround users do can translate into design requirements for better history tooling.

Chapter 2. Exploratory Programming as a Phenomenon

Research done in collaboration with Brad A. Myers¹

INTRODUCTION

In coding and in life in general, exploration is a basic human strategy for gaining new understanding about a space of ideas. In a seminal paper on organizational learning, March [March 1991] argued for a healthy balance between exploitation, the use of familiar knowledge, and exploration:

“Exploration includes things captured by terms such as search, variation, risk taking, experimentation, play, flexibility, discovery, innovation. Exploitation includes such things as refinement, choice, production, efficiency, selection, implementation, execution.... Maintaining an appropriate balance between exploration and exploitation is a primary factor in system survival and prosperity.” [March 1991]

The term “Exploratory Programming” was first popularized in a 1983 paper by Beau Shiel, a manager at Xerox’s AI Systems, who struggled with applying rigid software development lifecycles of the time to experimental AI code [Shiel 1983]. The trouble was that something so experimental as an AI system could not be fully specified up-front: *“no amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what works”* [Shiel 1983]. Before taking the long and expensive step of building software, it is generally recommended to rapidly iterate ideas through discussions and paper prototypes [Buxton 2010]. A simple paper-prototype of a user-interface can be shown to users to cheaply test the proposed design before any code is written [Buxton 2010]. However, it is not always feasible to test some ideas on paper [Yang et al. 2019]. When an idea relies on processing data, or on computing a complex visual, sound, or motion effect, these behaviors can be difficult or even impossible to simulate in “low-fi” prototyping mediums like paper or a whiteboard. Exploratory programming fills the crucial role of the medium when prototyping must occur in code. This practice is key for situations where key attributes of exploration: *“flexibility, discovery, and innovation”* [March 1991] are needed to understand how the program should behave.

At first glance, exploratory programming may seem difficult to separate from normal programming. Typical programming requires some experimentation and some creative problem-solving to reach a goal [Green 1990, Ko & Myers 2008]. However, the practice of designing the goal at the same time as experimenting in code is a defining feature of exploratory programming. We extend Shiel’s definition: *“the conscious intertwining of system*

¹ This chapter is based in part on the conference paper: Mary Beth Kery and Brad A. Myers, "Exploring Exploratory Programming," 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17), October 11–14, 2017, Raleigh, NC, pp. 25–29.

design and implementation” [Shiel 1983] to define exploratory programming as a task with two properties:

1. **The programmer writes code as a medium to prototype or experiment with different ideas.**
2. **The programmer is not just attempting to engineer working code to match a specification. The goal is open-ended, and evolves through the process of programming.**

To further gain an intuition for the boundaries of exploratory programming, consider debugging. Debugging often involves hypothesis testing and experimental code edits as a programmer tries to make sense of an error [Ko & Myers 2008]. Debugging thus meets Property 1. However, the broad space of debugging is not exploratory programming. Consider that a programmer knows how their program should be behaving, and finds an error. Through problem solving the programmer finds the erroneous lines of code, and through further experimentation, develops a fix. We do not label this as exploratory programming because at no point is the programmer re-evaluating their system design or how their program should behave (Property 2). Later, the programmer might decide that the system should perhaps behave in a different way, and then might engage in exploratory programming to try out new possible behaviors.

Our observations are partly motivated by our prior studies of exploratory programming [Kery et al., 2017], where we interviewed 10 researchers (2 female, 8 male) who self-identified as doing exploratory programming, followed by a broader online survey which received 60 responses from data scientists. Briefly, participants emphasized the trial-and-error nature of their exploratory programming work, and they reported using a variety of simple methods such as duplicating files, duplicating code snippets, or commenting out code snippets in order to keep multiple versions of the same code visible at once. Full results are in Chapter 3.

In this chapter we seek to clarify and deepen the understanding of exploratory programming, by synthesizing evidence of how exploratory programming is used in the wild and how exploration affects a programmer’s behavior. Today, we see highly exploratory code tasks, such as data science, learning through play, and computational art and design which need appropriate tool support, not only for professional developers but also for the broader end-user developer (EUD) audience. For example, programmers have a need to work with alternative versions and variants of their exploratory code [Hartmann et al., 2008, Kery et al., 2017]. Our aim is to fuel future research in exploratory programming by grounding the practice in 5 characteristics. These characteristics are distilled from data from our own prior studies [Kery et al., 2017], as well as prior literature. These are discussed at length in later sections of this chapter, along with related terms:

- A. **Needs for Exploration:** Certain scenarios call for exploration. This includes learning how to do an unfamiliar task, working on creative tasks, or working on hard tasks where the means to achieving a goal is not apparent without experimentation.
- B. **Code Quality Tradeoffs:** Exploratory programming emphasizes iteration on the ideas behind the code, so code quality is often deemphasized during exploration to allow for

- faster iteration. When the programmer reaches a final solution, they may then polish and refine the code.
- C. **Ease or Difficulty of Exploration:** Usability factors in the languages, libraries, and tools that the programmers use affect a programmer's ability to rapidly prototype. From Green's cognitive dimensions [Green & Petre 1996], we identify that high Closeness of Mapping and low Viscosity are particularly helpful for exploration.
 - D. **Exploration Process:** Exploration is instantiated as repeated changes to parameters, input data, or certain regions of code over time. This process also includes backtracking and comparing current code to past attempts to decide what ideas to experiment with next.
 - E. **Group or Individual Exploration:** Exploratory programming is often done on an individual basis, but can become highly convoluted when a team needs to coordinate their experimentation.

RELATED TERMS

Bergström and Blackwell discussed a number of programming practices, framing code as a medium with a great many other uses than typical software engineering work [Bergström & Blackwell 2016]. Some of these creative practices they discuss: bricolage, tinkering, sketching, live coding, and hacking, we consider to be a subset of exploratory programming. What makes “exploratory programming” a useful framing across many programming practices? From bricolage to hacking, as well as other practices we discuss below, a common trait is that the programmer’s goal is at least to some degree creative and open-ended. By examining exploratory programming, we can study the consequences on programmer behavior of working towards an open-ended goal, as well as tools that may benefit all of these practices that rely on goal exploration. Below we define several other relevant terms:

Opportunistic programming

As defined by Brandt et al.:

“Programmers build software from scratch using high-level tools, often add new functionality via copy-and-paste, [and] iterate more rapidly than in traditional development...” [Brandt et al., 2008]

Past work on opportunistic programming has focused on web foraging as a way that programmers rapidly iterate on their ideas in code, by largely patching together a program from online examples. As the programmer is often exploring different possibilities of their program, Opportunistic Programming can be considered a subset of exploratory programming. However, there are behaviors specific to this practice that do not strictly generalize to the broad range of all exploratory programming tasks. Patching-together example code is one exploratory programming tactic that may be specific to Opportunistic Programming.

Debugging into existence

Rosson and Carroll [Rosson& Carroll 1993] observed that Smalltalk programmers write partial code and run it, so that the resulting errors could point them towards where to improve the program. This style of programming is highly incremental and can be used for exploratory programming, if the programmer has an open-ended goal.

Rapid Prototyping

“*The rapid production of a prototype*” [Oxford English Dictionary] is commonly used in iterative design to create and test a number of different design possibilities early on. Note that rapid prototyping may or may not involve programming, and the user may have a specific goal and design in mind. Rapid prototyping in code is exploratory programming if the programmer is exploring a variety of designs/goals rather than simply iterating on a single design.

CHARACTERISTICS OF EXPLORATORY PROGRAMMING

Needs for Exploratory Programming

Exploratory programming has been observed and purposefully supported in a wide variety of applications which fundamentally require exploration, including:

- **Learning programming through play:** Environments for children such as Alice [Kelleher et al., 2007] and Scratch [Resnick et al., 2009] encourage creating stories through exploratory programming.
- **Digital art and music:** Digital art written in languages like Processing is created through experimentation with code as a creative medium [Reas & Fry 2007, Montfort 2016]. Environments for generative music, often using live coding, involve impromptu exploration of sounds through code [Brown & Sorensen 2009].
- **Data Science:** Tasks like data analysis are often done in code and are exploratory [Tukey 1977]. Other tasks, like modeling or building a machine-learning model, can also take extensive exploration and iteration in code [Hill et al., 2016].
- **Software Engineering:** Exploratory programming has been found in programmer’s backtracking, where the programmer tries and retries different alternatives using commenting or undo commands while trying to determine an appropriate algorithm [Yoon and Myers 2014] or figuring out how an API should be used [Robillard 2009].

Code Quality Tradeoffs

Programmers frequently need to make a cost/benefit trade-off between producing high-quality code, and spending their time and effort on quick ideation. Historically, this has meant that exploratory programming is associated with rough code: “Exploratory programming techniques encourage code that is hard to read. It is tempting to make fix after fix to a piece of code until it is impossible to understand. Hence, rewriting code is essential for producing reusable code” [Sandberg 1988].

In our interview study (Chapter 3), even participants who had formal software engineering backgrounds leaned towards messier practices when exploring. When participants were writing code to be part of a maintained software system, or were simply more meticulous, they sometimes strived to follow good coding practices. However, all participants mentioned some way in which they preferred to reduce engineering effort while writing exploratory code, whether ignoring modularity, skipping documentation, or avoiding software version control. Some participants felt they wouldn’t use this code long-term. Others did not want to invest too much time on code that may turn out to be a bad avenue that would need to be re-written later anyway.

In a study of game developers, Murphy-Hill et al. had similar findings [Murphy-Hill et al., 2014]. Game development often requires exploratory programming due to evolving requirements and the creative nature of building a game. Even though subjects were professional game developers, they expressed this quality trade-off: “there is a tradeoff between improving maintainability early and the likelihood that this effort will result in waste because the game will not be a success” [Murphy-Hill et al., 2014]. Programmers may choose low-investment in their code quality due to time pressures and risks of their immediate work later being thrown out for a new idea. Yet exploratory code faces the same problems with bugs and logic errors as any program. Poor quality exploratory code can lead to serious problems, such as incorrect data analyses or invalid scientific findings [Merali 2010].

Ease or Difficulty of Exploration

In the course of exploration, a programmer may need to significantly edit the design of their program to try a new approach. All programming tasks, including exploratory changes, can be made easier or more difficult by the tools available to the programmer. Here we use some of Green’s cognitive dimensions of programming languages [Green & Petre 1996] to discuss particular usability features that support exploration.

First, languages that cost a programmer more time and effort to express a single idea will slow down iteration. This relates to Green’s notion of Diffuseness versus Terseness of a language, which counts how many code symbols are required to express an idea. For exploratory programming, high-level languages and libraries can be extremely helpful to provide a higher-level vocabulary to make code more succinct. For instance, instead of coding the details of a computer vision algorithm to detect contours in an image, a programmer can simply use a one-line library call such as `find_contours()`².

Green’s Closeness of Mapping is how directly a concept in the user’s task domain maps to a code representation [Green & Petre 1996]. Good closeness of mapping for exploratory programming will favor higher-level abstractions for all parts of a program that are not the programmer’s primary focus. For example, `find_contours()` allows a programmer to use computer vision without needing to understand its low-level algorithms. Green called side tasks that only relate to programming, such as explicit memory management, “programming games” [Green & Petre 1996]. We expand the “programming games” notion to encompass any supporting domain that a programmer may want to make use of without having to care about the details of that domain. The overall goal of close mapping and terse code is to help an exploratory programmer spend more time and effort focused on their ideas rather than the details required to enable them. Here, exploratory programming has certain overlap with opportunistic programming, and is farther from standard software development which prioritizes engineering goals such as efficiency of execution, flexibility, maintainability, and robustness over easy implementation. Close mapping also makes exploration more accessible to novices and end-user programmers who may be missing certain skills. For experts, Closeness of Mapping also implies that abstraction should be as fine or coarse-grain as is reasonable for the task. For example, if a domain expert is exploring new algorithms for computer vision, a very low level of abstraction may be appropriate.

² scikit-image: image processing in Python <http://scikit-image.org/>

Viscosity [Green & Petre 1996] refers to how easy or difficult it is to make a change in a program once it is written: “programmers will choose their style of working according to the particular combination of information structure and editing tools. If the system is viscous, they will attempt to avoid local changes and will therefore avoid exploratory programming” [Green 1990]. To avoid viscosity, highly modular programs may help programmers explore a single component without needing to propagate changes across the entire system to make that change run. High viscosity can also lead to errors. For example, an interview participant from our Exploratory Programming Study [Chapter 3] discussed errors that resulted from re-using the same block of code for multiple experiments. Editing the code for one exploration made it difficult to maintain or recover an earlier exploration. Here, viscosity must take into account how easy it is both to create a change and to revert it. Exploratory programming environments should lower the risk of making exploratory edits by providing clear ways to return to prior versions. A simple undo is not sufficient when programmers make changes over time that they would like to only partially revert [Yoon and Myers 2014]. Prior work has found end-user programmers, even in visual programming, benefit from more sophisticated version control support to help with reverting [Kuttal et al., 2011].

EXPLORATION PROCESS

The process by which exploratory programming is done can be characterized by backtracking, the scale and duration of the changes, and history.

Backtracking

Yoon et al. [Yoon and Myers 2014] examined 1,460 hours of programming log data from 21 programmers. Yoon identified evidence of exploratory programming when a programmer edited the same piece of code over and over between runs, sometimes *backtracking* to an earlier iteration of the code. Following Yoon’s convention, we identify an instance of exploration as two or more edit-run cycles that are close in time and affect the same code. Segmenting exploratory changes to code by runs may often be appropriate, as a programmer is experimenting with changes to the program, and must typically run the code in order to see the effect of those changes.

Exploration scale

Identifying the scale of an exploratory section of code is helpful, as scale affects how a programmer will interact with the variants of that code. In conventional software version control tools, a set of changes on a source file is captured at the file level. However, in a practical exploratory experiment, the programmer may be manipulating a much more focused set of code than the entire file. At the smallest scale, a well-documented behavior is “tuning” a single variable or parameter to many different values to observe the effect [Yoon and Myers 2014, Snoek et al., 2012]. An artist, for instance, may change a parameter in a program that generates a complex geometric shape [Terry et al., 2004]. At the next largest scale, a programmer may be rapidly iterating variations of a particular function. For instance, one of our interview participants created two copies of the same function to try two different approaches to an analysis and still keep both approaches. At the next larger scale are loose snippets of code that span multiple functions, or are not contained by a function at all. For example, one interview participant ran a different “configuration” of their file by commenting out certain lines across

their file and uncommenting others to change what analysis was performed. Finally, exploration may also occur at the file level.

Exploration duration

As a programmer shifts among different tasks, the exploration scale may vary not only by code size, but also by time spent. Exploration may be very transient or very long-term, depending on the task. For instance, if a programmer is changing the color and size of a button, this may be only of concern while the programmer is deciding which color and size is liked best. This contrasts with cases of building computational models where the programmer is involved in exploratory programming for weeks to months and must keep track of many explorations that make up the components of their model [Hill et al., 2016]. Relating to Green's cognitive dimension of hard mental operations, the burden on a programmer will be greater for keeping track of many attempts over a long period of time than keeping track of fewer code variations over a short period of time [Green & Petre 1996]. All of our interview participants used notes, code comments, or recorded output as external memory aids. However, in line with Code Quality (Characteristic B above), most noted a high cost in effort to keep these manual records up-to-date.

Using Exploratory History

In software engineering, history involves byproducts of the process such as code versions, commit messages, and issue logs, as well as the code changes themselves. Prior work has found that software engineers typically use code history to understand a change or bug [Codoban et al., 2015]. On the exploratory end of the spectrum, exploratory programmers may use history for typical software engineering needs, but they also commonly use code history as a record of their experimentation. Abundant evidence for this is seen in scientific computing [Davidson & Freire 2008]. Whereas a software engineer might ask a code-centered question like: “*In which version did this code appear?*” [LaToza & Myers 2010], an exploratory programmer may frame their question around an experiment: “*When I tried a RandomForest algorithm, how did that affect my model’s accuracy?*” or “*When I went through this loop 3 times, how did this affect the character’s motion?*”. Using history for this kind of question and answer was mentioned by several interview participants. To facilitate answering these questions, a programmer during exploration may need a variety of artifacts that can help them understand a past decision and its effect. This includes past versions of images, notes, variable values, parameters, and graphs, along with the code. In the Exploratory Programming Study (Chapter 3), we saw participants keeping versions of code, jotting notes on ideas, and keeping versions of outputs. Of survey participants, 72% manually copied versions of their files and 3 out of 10 interviewees and 52% of survey participants used a software version control tool.

Sharing and Group Exploration

Group exploration on the same code can be challenging because the kind of informal coding practices that appear in exploratory programming do not lend well to clear code. Due to programmers’ reluctance to keep up-to-date notes during exploratory tasks, keeping a shared understanding of an exploration’s progress across a team can be difficult. Sharing and group exploration also affects the viscosity of code edits, because where more than one person is

making exploratory changes to the source code, this can easily lead to interfering changes. Wang et al. observed many of these kinds of coordination and communication difficulties in their study of data scientists synchronously editing the same computational notebook together [Wang et al. 2019]. Based on their data Wang et al. suggest future work focus on facilitating communication, such as through chats, or better hybrid management of which parts of an exploration are “owned” by an individual author versus shared between collaborating authors.

CHAPTER CONCLUSIONS

Exploratory programming is a practice and a lens we can use to better understand and support creative and open-ended programming tasks. Although exploratory programming is prevalent across many applications today, there is currently a lack of tool support for experimentation, including a lack of support for recording and sensemaking of exploration history, and a lack of support for exploration by groups of people. Next in this dissertation we further investigate specific practices of exploratory programming for data work and start the design process of understanding how we might better support exploration history.

Chapter 3. Informal Versioning Behaviors in Data-centric Exploratory Programming

Research done in collaboration with Brad A. Myers and Amber Horvath³

INTRODUCTION

People have been using code to analyze data since the dawn of computing, and so despite its surge in popularity in the last decade, one might think that exploratory programming with data would have long-established best practices. This is not entirely the case. It turns out that *keeping track of ideas* in experimental work remains an unsolved problem. Patel et al. observed that many difficulties in applying machine learning techniques arose from the “*iterative and exploratory process*” of using statistics as a software development tool [Patel et al., 2008]. Patel’s interviews with machine learning developers emphasized the non-linear progression of this work where “*an apparent dead end for a project was overcome by revisiting an earlier point in their process*” [Patel et al., 2008]. Others who have studied data science and machine learning developers, such as Hill [Hill et al., 2016] and Guo [Guo 2012] have described difficulties even for experts, struggling to create understandable and reproducible models during a process where they attempt many different things. Both Hill and Patel called for advances in software engineering methods and tools for dealing with this kind of programming. Similar arguments have been made in the scientific computing community, where problems of understandability and reproducibility during experimentation with code are often mentioned [Segal 2007, Nguyen-Hoan et al., 2010, Merali 2010]. When even experts struggle with the exploratory process, this lowers the accessibility to novices programming with data.

In this research, we sat down with data scientists using a combination of interviews and looking at their code artifacts, followed by a broader survey, to better understand the barriers and requirements of data scientists managing exploratory code.

METHODOLOGY

Interview Study

We conducted a series of semi-structured interviews with researchers across multiple universities. Researchers were a convenience sample of our target population: people who do significant exploratory work with data. We recruited individuals who had worked on at least one major exploratory analysis project. Our 10 respondents were a mix of faculty, graduate and undergraduate student researchers. Eight of interviewees did research in a computer science-related field, one in computational chemistry, and one in computational neuroscience. The gender ratio was 2 females to 8 males. Interviewees worked with a variety of programming

³ This chapter is based in part on the conference paper: Mary Beth Kery, Amber Horvath, and Brad A. Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In CHI, vol. 10, pp. 1265–1276. 2017.

languages, with Python and R being the most used. We intentionally oversampled people who were experienced programmers with computer science training in order to better understand the intent of their practices not simply arising from lack of awareness of available tools or lack of skill with software development. Prior work has shown that many scientists are unaware of good software development practices [Nguyen-Hoan et al., 2010].

All participants first signed a consent form. There was no monetary compensation for participation in the study, and participants volunteered their time for a 45–60 minute interview. In the first part of the interview, we asked participants to describe a recent exploratory project at a high level: What were their goals in the project? What high-level steps did they follow to meet those goals? All interviewees discussed projects that had spanned at least several weeks of work. Due to the timespan of significant projects, we chose a retrospective interview methodology rather than direct observation of their work [Brandt et al., 2008]. Next, we asked participants, whenever possible, to show us artifacts from the project, including source code, their folder structure, and data files. During this stage we asked participants to discuss how their high-level ideas had been implemented in code and files.

Each interview was audio recorded and transcribed. Several participants gave us permission to keep and share screenshots of their code and files, and these artifacts were used in our analysis. As the interviews were focused on each participant’s research, there were large parts of the transcriptions about an interviewee’s general research topic, rather than their work process. To analyze the interviews, two coders first read all interviews and pulled out any quotes related to the process, such as plans, code, notes, collaborators, etc. Following an affinity diagramming approach, coders grouped the quotes into higher-level themes, and separated out any quotes that explicitly mentioned a difficulty or complaint.

Survey

We next sought to validate our observations from the interviews on a broader population. Using an online survey, we recruited respondents from several websites for data scientists (e.g., kaggle.com and reddit.com groups for machine learning or data science), as well as emails to acquaintances. A total of 77 people started the survey. However, not all participants answered all of the questions, so here we analyze only the 60 people who answered the questions beyond the demographic information. All 60 self-identified as having experience coding with data in an exploratory way. The average age of participants was 34 (SD = 13), and the gender ratio was 21% females, 74% males. The remaining 5% of participants chose not to disclose their gender.

We structured our survey such that it acted as a quantitative supplement to our interview. Using the interview results, we drafted questions that built upon what issues affected participants most, what design features in a tool they would want to address these concerns, and to see if our findings generalize across a more diverse sample. First, we asked questions to determine the background of our participants, summarized in Figure 3.1. We then asked questions about coding practices and behaviors. We presented statements such as “I analyze a lot of different questions about the data in a single source file” with a 5-point Likert scale, going from “Never” to “Very Often”. We then asked about the problems they encountered, such as “Distinguishing between similarly named versions of code files or output files” with a 5-point Likert scale going from “Not at all a problem” to “A very big problem”. We also gave a “don’t know / can’t

“answer” option in case participants had never encountered or could not recall encountering such an issue.

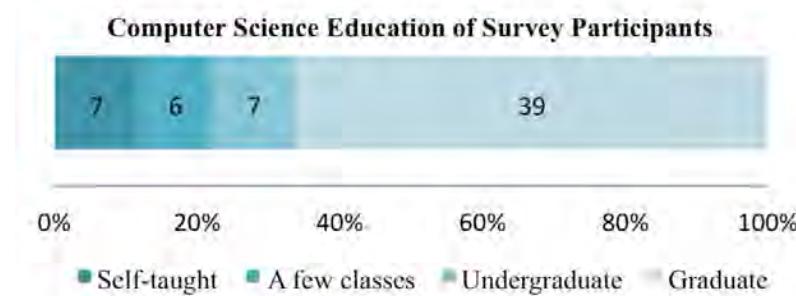


Figure 3.1. Background statistics about 59 of the survey respondents. Most had graduate degrees in computer science and work in research (one respondent declined to answer).

RESULTS AND DISCUSSION

Interview and survey participants varied widely in their practices, and the kinds of projects they worked on. Despite this, many participants had behaviors and beliefs in common. In the following, participants in the interviews are identified with a “P”, and survey responses with percentages for the different answers.

Exploratory Process

The participants mentioned a variety of ways that their programming tasks were exploratory. One of the most salient feelings expressed by participants was the trial-and-error nature of their work, and the risk of investing in an idea that may fail or be discarded:

“I didn't always have a great idea of what would work up front, so I would try a lot of different things and then they wouldn't pan out and then you would disregard most of that work, but maybe still want some of the small processing steps from that work, if that makes sense, to apply to your next statistical model.” – P10

Of survey respondents, 43% “Agreed” on a Likert scale that they “prioritize finding a solution over writing high-quality code”, while another 33% “Strongly Agreed” (totaling 76%). Although some interviewees were distinctly more messy or meticulous as evidenced by their code artifacts, all mentioned avoiding investment in some way, whether avoiding leaving informational comments in their code, or avoiding taking notes or not using extra software tools beyond the bare minimum needed for their analysis.

“I know how to write code. And I know that I could write functions to reuse functions and I could try to modularize things better, and sometimes I just don't care because why am I going to put effort in that if I'm not going to use it again?” – Po6

This sentiment is common to how end-user developers prioritize goals [Ko et al., 2011]. While “end-user developer” often refers to a programmer without formal training in computer science, many of our participants did have formal training (Figure 3.1). Ko et al. distinguish

end-user developers as having goals where a program is a means to an end, rather than professional developers, whose goals are the code itself as a product [Ko et al., 2011]. Under this definition, considering data scientists as end-user developers may be fruitful for leveraging existing theories on programmers who write expendable code.

Yet such “throw away” data analysis attempts are often not really thrown away. While all interviewees discussed accumulated failed attempts and earlier analyses that were less informative, they also often talked about reusing that code. Interviewees mentioned they often built off code from an earlier attempt in order to try a new method. This is supported by the survey, where 46% of survey participants reported reusing code from the *same* project at least “Often”. Similarly, 47% reported reusing code snippets taken from *different* projects at least “Often”.

Informal Versioning

Data scientists we interviewed and surveyed faced challenges of trying out multiple alternatives in their code, while trying to judge which code to keep in case that analysis or helper method would be useful again later. Exploration can involve nonlinear iteration, so keeping code to backtrack to or to reuse was important to interviewees. 4 of 10 participants discussed actively keeping around old code they were no longer using, just in case some part of that code proved helpful later on. Similarly, 65% of survey respondents reported leaving code snippets they were not currently using in the code at least “Occasionally”, and 79% reported commenting out code at least “Often”.

Interviewees were cautious about deleting code. Yet this introduced code complexity, as some attempts could not simultaneously exist in the same namespace, or used overlapping code. As quick workarounds, data scientists relied on informal versioning such as commenting:

“I guess this is kind of my own personal version of version control. A lot of times I'll, like, comment out a whole big section that was there, and then I'll rewrite it so that it's different but I'll keep the original one exactly as is in case the new version kind of sucks.” – P09

Code that is commented out does not run. Using comments to store code has been observed in prior studies [Hartmann et al., 2008, Segal 2007]. Ko et al. [Ko et al. 2005], when studying experienced programmers, found that 60% of edits using comments were for temporarily commenting code during maintenance tasks.

Comments to keep track of attempts

44% of survey takers reported using comments to keep track of what they have tried, and 70% keep commented code to reuse later. This allowed data scientists to keep multiple versions of an idea for reference, with only the relevant one running.

Comments to manipulate execution

Comments were used not only to store chunks of code, but also to mutate the meaning of existing code, sometimes in complex ways. In the survey, 56% of the respondents reported using comments explicitly to control execution.

P03's code, shown in Figure 3.2, shows an example of this, in which several alternatives for outputting the analysis result are present in the comments. An active chunk of Python code graphs the output, but this code has several commented-out statements. There is also a second graphing section of code lower down that is fully commented out.

```

537 print "hc policy distribution entropy: \n" + str
538 #print "2-fold CV Var: \n" + str(CV_variances)
539
540 #print "Stdev of (V_M under h^f, avg over states
541 #print "Stdev of (V_M under h^c, avg over states
542 #print "Average V_M under h^f: \n" + str(V_M_hf)
543 #print "Average V_M under h^c: \n" + str(V_M_hc)
544
545 fig = plt.figure(figsize=(9,4))
546 ax = fig.add_subplot(1,1,1)
547 ax.set_xticks(D_sizes)
548 plt.plot(D_sizes, [V_star_avgState]*len(D_sizes)
549 #plt.plot(D_sizes, AvgState_V_hfs)
550 plt.errorbar(D_sizes,AvgState_V_hfs,yerr=Std_Vs_
551 #plt.plot(D_sizes, AvgState_V_hcs)
552 plt.errorbar(D_sizes,AvgState_V_hcs,yerr=Std_Vs_
553 plt.xlim(0,max(D_sizes))
554 plt.xlabel('|D|')
555 plt.ylabel('mean(V)')
556 plt.grid()
557 plt.show()
558 ...
559 ...
560 fig = plt.figure(figsize=(10,4))
561 ax = fig.add_subplot(1,1,1)
562 ax.set_xticks(D_sizes)

```

Figure 3.2. Comments used to keep alternatives. Above, `print` statements are commented out using the single-line comment token `#`. Each prints out a different statistic about the data. The one `print` statement on line 537 remains uncommented, meaning that statistic will be outputted when the program is run, and that statistic is the one P03 is currently interested in seeing. Similarly, comments on lines 549 and 551 toggle what data is plotted in P03's plot. The block comment syntax on lines 559–562 toggle a different plot, that P03 has currently turned off.

Duplicating snippets, function and files

Copying code was another popular way of versioning. Shown in Figure 3.3 is P01's file structure, where many versions of the same script were kept to track major attempts at improving a machine learning model. On average, 72% of the respondents in the survey said they at least “Occasionally” do this, and 58% at least “Often” said they named the new file copies based on the original one. Survey respondents reported making an average of 3–4 versions based on one file. Interviewees also demonstrated multiple versions and copies of functions and smaller code snippets.

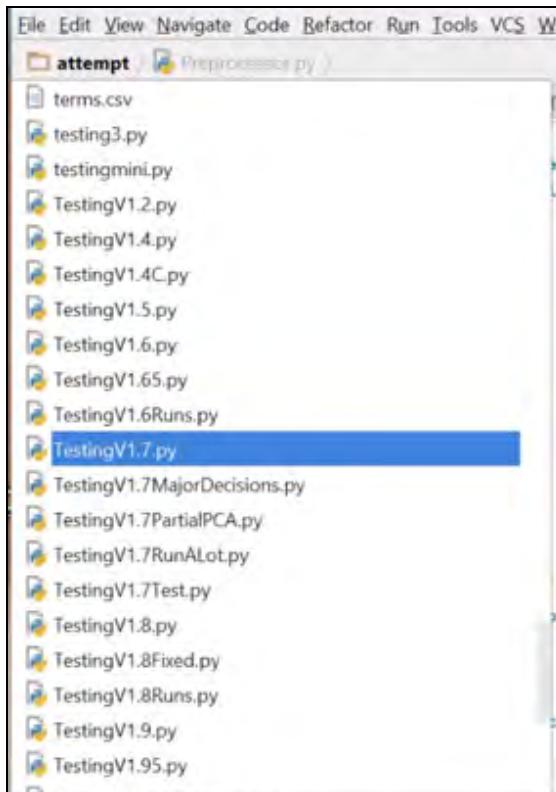


Figure 3.3. Folder from a data analysis project of P01

Difficulties

As interviewees did not overly invest in notes, comments, or trying to write clear code more than they felt necessary, they had to rely on their mental map of their code to understand it. Here are some difficulties that interviewees mentioned, and responses from the survey that show that these issues are indeed widespread:

Why did I name my file that?

Participants discussed having files or methods with ambiguous names or that they often had multiple files with similar names, making it difficult to distinguish between versions (see Figure 3.3). 7 out of 10 participants expressed confusion when talking about the names they chose for different methods and files. 83% of survey takers reported that closely named artifacts had caused them at least minor problems in their work.

How do I keep track of everything in my project?

Participants struggled to keep track of the relationships between files (source code, input data, output data), code snippets, and their analysis progress. 5 out of 10 participants expressed having difficulty in keeping track of the high-level aspects of a project and how it related to the lower-level code. Furthermore, as their code evolved, the code that originally produced a particular result may be changed or obfuscated. 4 out of 10 reported losing track of their mental map of code, especially if the code was messier or had parts commented out. Interviewees discussed understanding their code in the short term, but having trouble understanding the

code when they returned to it later. 85% of survey takers reported that this caused them at least minor problems in their work, with 44% reporting significant problems.

What was I doing in this old project?

Participants wanted to go back to old projects to lift code, but had difficulty remembering and understanding the structure and details of these old projects. 5 out of 10 participants expressed difficulty reorienting themselves with older projects. 67% of survey takers reported that visiting old projects was a significant problem.

What do I do with all this old code?

Participants expressed an interest in “hoarding” code through commenting out code snippets and refusing to delete old source code files in case their exploration did not pay off. However, this led to confusion with keeping track of multiple similar copies as well as commented versus not-commented code.

This file is huge! What's in it?

A script file is a kind of code file that can be directly run/executed and typically directly outputs print results or file results. Participants often had large script files that served a variety of purposes, resulting in confusing code dependencies and relationships among various parts of the script. This resulted in an overall difficulty discerning the purpose of a script, which 4 of the 10 participants mentioned. Some reported the problem extending across multiple files, where cluttered directories were composed of confusing relationships between files.

These alternatives are inconsistent!

P01 faced problems in which she fixed a bug in one alternative of her code, and then when she needed to backtrack to an earlier alternative, she had lost track of which alternatives had the bug fix, and which did not. 74% of survey takers reported that inconsistent alternatives had caused them at least minor problems, with 35% reporting that this was a significant problem.

Why aren't people using version control systems?

Data scientists face difficulties with informal versioning for multiple overlapping ideas over time. This might seem to be well within the realm of problems that software version control systems (VCSs) are designed to solve. VCSs reduce code clutter by separating out versions. They give order to versions and preserve history, so that older versions and analyses can be reproduced in the future.

However, only 3 out of 10 of interviewees chose to use software version control for their exploratory analysis work, even though 9 of 10 did actively use VCSs such as Git or SVN for other non-exploratory projects they worked on. A benefit of over-sampling from individuals with a computer science background in the interview study is that this provides a more nuanced picture than previous work, which has studied version control usage by scientists who lack training in computer science [Nguyen-Hoan et al., 2010]. Nguyen-Hoan et al., in a survey of scientists, found that 30% of their sample used VCS [Nguyen-Hoan et al., 2010]. Despite our oversampling of people with CS backgrounds in our survey (see Figure 3.1), we found that 48%

of participants, just under a half, did *not* use version control for their exploratory data analysis work.

A few survey takers did not know how to use software version control, but the most common reasons for not using a VCS were 1) it was too heavy-weight for what they needed, 2) they were not concerned about code collaboration, and 3) they were not concerned about reverting code. This final reason “No need to revert code” was cited by interviewees as well, and appears contradictory to their demonstrated interest in hoarding old code for later reuse. In fact, some interviewees, when pressed to explain, reasoned that because their code was copied and commented in many places, there was no need to “backtrack” or “revert” in the sense that all old code they needed was present in one of their code files.

“I guess because I’m doing like this copy and pasting thing, I also have lots of old versions of stuff everywhere, like other projects and things like that” – P05

It is clear that many data scientists do not perceive VCS as having enough benefit over their current practices to invest effort in using these tools. Furthermore, while using informal versioning can be problematic, we argue there is functionality of these interactions that conventional VCS does not gracefully support:

- Versions are easily accessible and comparable because they are all available in the user’s immediate files.
- It is easy to see what code you have available to reuse.
- There is a smaller learning curve, since this should not be much more complicated than the commands it takes to comment or copy something.
- It is easy to temporarily create a version of the code, and then remove the version if not wanted later.
- It is easy to keep alternatives of an arbitrary size. While conventional version control operates only at the file level, programmers make use of commenting and copying to version at the level of functions, code snippets, lines, or even single values.

In order to make exploratory programming less prone to confusion, we aim to inform new interactions for software version control tools based on how data scientists naturally use versioning.

CHAPTER CONCLUSIONS

The study described in this chapter contains key hints about what makes history for exploratory programming *hard* and what better history tooling would need to resolve those pain points:

1. First, participants’ commenting behavior suggests that **content smaller than an entire code file should be considered for versioning**. These smaller snippets contain important content like plots, models, and individual analyses.
2. Second, participants’ copying and commenting behavior suggests that it is valuable for practitioners to have **easy access to multiple versions of the same thing in their workspace**. This is a key difference from conventional version control where a programmer would have just their single immediately previous version to compare to.

3. Third, participants faced a lot of difficulty remembering which versions were which, especially after a long time when they needed to re-orient themselves to an older project or file. History support will need to **help users understand what content is in the versions they have and tell the difference between similar versions.**

Finally, it should be acknowledged that some of the difficulties faced by practitioners in this chapter may not be fixed directly by history support. For instance, in “*This file is huge! What’s in it?*” participants had enormous script code files that were difficult to navigate. While history tooling may not assist that problem directly, we hope that practitioners will not feel the need to keep so much clutter within their files or folder structures, if they know that they can rely on good history support to manage that alternative content for them.

Chapter 4. Usage of Notebooks in Data-centric Exploratory Programming

Research done in collaboration with Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers⁴

INTRODUCTION

As discussed in Chapter 3, currently even experts struggle to keep track of the experimentation they do. Insufficient documentation of data work over time can lead to lost work, confusion over how a result was achieved, and difficulties effectively ideating [Hill et al., 2016]. Literate programming has recently arisen as a promising direction to address some of these problems [Knuth 1984]. It originates from a 1984 paper by Donald Knuth:

“Time is ripe for significantly better documentation of programs, and that we can achieve this best by considering programs to be works of literature.” [Knuth 1984:1]

Knuth’s philosophy held that humans should write code foremost as a natural language prose expression of their reasoning, which a literate programming tool facilitates by allowing formatted text annotations to be rendered inline with plain code. The actual computer code would be a secondary translation of these essay-like annotations [Knuth 1984]. Knuth’s intended audience, mainstream software engineering, largely did not adopt the idea. Arguably, literate programming is a poor fit to software engineering, as documentation quickly gets out of date and is costly to maintain [Lethbridge et al., 2003, Parnas 1994]. Heavy annotations also conflict with the idea in software engineering that well-structured code should “speak for itself” and be understandable with minimal documentation [Martin 2009].

Yet in a different community, programming for math and sciences, literate programming has thrived since 1988 in tools such as Mathematica⁵. For a biologist, physicist or financial analyst who codes an analysis, there may be theories and equations embodied in their source code which could benefit from additional explanation as formatted equations or images [Xie 2014]. By allowing the mix of any media needed to understand domain-rich code, literate programming has gained an important role in the sharing and reproducibility of computational analyses [Xie 2014, Kluyver et al., 2016]. Today, open source literate programming tools like Jupyter notebooks and knitr⁶ have become hugely popular, with millions of users across a wide range of expertise and subject domains.

⁴ This chapter is based in part on the conference paper: Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. "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, pp. 1-11. 2018.

⁵ Wolfram Research Inc. Mathematica, <https://www.wolfram.com/mathematica/>

⁶ Knitr tool, <http://yihui.name/knitr/>

Although literate programming currently aids a programmer to communicate an analysis [Shen 2014], data science tasks are known to be highly iterative and exploratory [Shen 2014]. For every useful model feature or insightful visualization data scientists create, there may be many less-successful features, plots, or analyses they have tried. In order for the original programmer or another person to improve and build upon their work, knowledge of this exploration is important. Data scientists need to keep track of what they have attempted, including failed approaches, to justify why they choose certain approaches over others and to be more effective in their ideation [Hill et al., 2016]. Current approaches, including traditional version control, have been found to be ineffective or require high amounts of tedious manual note taking on part of the data scientist (Chapter 3). With the increased ability to keep code, data, input, and output together in one document, we wondered if literate programming helps data scientists retain the story of their in-progress exploration. Some in the scientific computing literature say yes. An article in Nature says that a literate programming tool like Jupyter notebooks “helps researchers to keep a detailed lab notebook for their computational work” [Shen 2014]. However, although literate programming is an essential and distinct kind of programming in data computing today, there have been no studies on how literate programming affects programmers. In this chapter we present two studies that investigate how data scientists explore ideas as they develop code, how and why they develop a narrative structure in a literate programming tool.

In our first study, we interviewed 21 professional data scientists who use Jupyter notebooks, a popular literate programming tool in data science with over 2 million users as of 2015 [Perez & Granger 2015]. We found that although notebooks are limited in how they can be used to keep a detailed record of all explorations, several patterns of curatorial behavior emerged during iteration to build narrative.

In a second study, we probed how future tools may improve data scientists’ interactions with exploration history. We surveyed 45 data scientists and asked how they might use historical records of their analyses if they had a magical oracle to deliver any prior analyses content to them. These results revealed highly varied ways of interacting with history. Finally, we discuss implications for future research.

BACKGROUND & RELATED WORK

Notebook programming environments

A “notebook” environment is one particular genre of literate programming tools that is supported by many data-centric literate programming tools such as Jupyter Notebooks, Mathematica⁷, Databricks⁸, Apache Zeppelin⁹, and Sage Notebooks¹⁰. Although here we specifically study Jupyter Notebooks, we report on the usage of core features like cell structure,

⁷ Wolfram Research Inc. Mathematica <https://www.wolfram.com/mathematica/>

⁸ Databricks (2018) <https://databricks.com/>

⁹ Apache Zeppelin <https://zeppelin.apache.org/>

¹⁰ The Sage Mathematics Software System had a notebook interface predating Jupyter Notebooks, appearing in 2005 <https://www.sagemath.org/index.html>

cell layout, and cell types, which are common features to this entire genre of tools and should allow our findings to generalize more broadly.

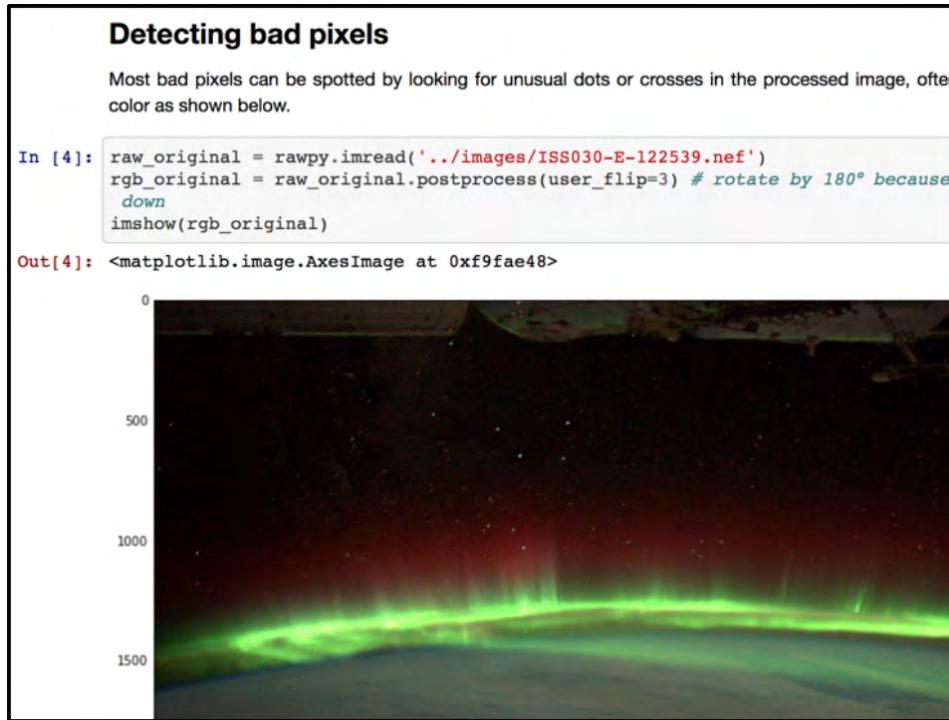


Figure 4.1. Excerpt from a Jupyter notebook¹¹

A notebook environment supports chunks of content, called “cells.” A cell can contain code, output, a table, a plot, formatted “Markdown” text, or other kinds of media. For example, the first cell in Figure 4.1 contains formatted Markdown text (indicated by no background color and no number in the left margin), the second cell contains python code (labeled In [4]:), and the third contains the textual and graphical output of that code, which is updated each time the code in that cell is run. With a notebook, a programmer can produce a literate program that fits Knuth’s ideal definition: a chronological progression of Markdown cells, code cells, and output cells, explaining everything from top to bottom. However, the notebook environment does not enforce this structure. A programmer is not forced to add Markdown explanations or any code comments. A notebook also allows each code cell to be edited and run individually at any time in any order, so rather than running the entire file from top to bottom, or only editing at the end of the notebook, programmers can pick and choose which code cells they would like to edit and run. As notebooks do not actually enforce a literate style, the programmers’ situational needs are likely what motivates them to create a coherent literary-style document or not.

Literate programming tool developers ask their users for feedback. For example, in 2015, the Jupyter Project surveyed over 1000 users on general usage, pain points, and feature requests [Jupyter Notebook UX Survey 2015]. They reported a quantitative analysis of all data, including

¹¹ Maik Riechert. 2016. Repairing Bad Pixels. Retrieved January 6, 2018 from <https://github.com/letmaik/rawpynotebooks/blob/master/bad-pixel-repair/bad-pixelrepair.ipynb>

keyword frequency counts on free-text responses. Version control was the most highly requested feature, although what the respondents meant by “version control” and what pain points it was intended to resolve was not probed. In addition to these data, there are many public examples of Jupyter notebooks online (for instance, a simple search on GitHub yields over 89,000 notebooks). However the artifacts themselves do not provide enough detail about small-grained iterations and intention [Yoon et al., 2013] to answer our research questions, requiring us to use different methods to understand how data scientists explore ideas.

Lab notebooks and scientific documentation

Lab notebooks are a log of scientific activity that contain enough detail for a scientist to later reproduce their experiments [Klokmos & Zander 2010]. Lab notebooks are ideally immutable logs once written, in order to serve as legal evidence of discovery in patent cases or questions of scientific validity [Klokmos & Zander 2010]. Lab notebooks take the form of paper, digital note-taking tools, and hybrids between paper and digital notes [Tabard et al., 2008]. Prior studies have explored the design needs that scientists have for their lab notebooks and their struggles with searching and maintaining scientific records [Klokmos & Zander 2010, Oleksik et al. 2014, Tabard et al. 2008]. Notebook programming, like any digital tool that can record text, can be used as a lab notebook [Shen 2014] but can also serve many other purposes. Prior studies have largely focused on wet-lab scientists, whereas a chief difference in our current behavioral study is our sample of computational analysts who primarily work through code. Thus our investigation is not just on the record keeping, but on the iteration of their primary work through code.

NOTEBOOK USAGE STUDY

Method

To get as unbiased a view as possible of data scientists working with a literate programming environment, we followed the Grounded Theory Method (GTM) described by Corbin and Strauss [Corbin & Strauss 1990] for data collection and analysis. We had the opportunity to collect data at the inaugural conference for Jupyter Notebook users (JupyterCon 2017). This provided a concentrated sample of people who have experience using Jupyter notebooks for real-world professional data analysis programming. Participants were recruited through conference speakers announcing the activity and organizers tweeting about it.

We interviewed 25 participants, but 4 were removed from our analysis because these interviewees turned out to be managers or otherwise did not personally do data analysis in notebooks. The final 21 interviewees held job titles shown in Table 4.1 and reported gender identity as 19 men and 2 women. Interviews lasted 10 to 30 minutes, based on the participant’s availability.

Table 4.1. Interview participants' primary job roles

Role	Participant
College teacher & researcher	N01, N02, N10, N15, N17
Financial analyst	N05, N06
Computational Biologist	N08
Software Developer	N19
Data visualization designer	N03, N04, N21
Data Scientist	N07, N09, N11, N12, N13, N14, N16, N18, N20

The interviews were planned around a few open-ended questions, beginning with an overview: “Please tell me briefly what you use Jupyter notebooks for?”. The next probe asked for details of what the interviewees did in notebooks to develop their ideas from inception to final result (over 60% of the interviewer’s utterances were on this topic). If participants had their laptop with them, they were invited to show the interviewer their actual notebook documents, and four of the participants did. Our other planned questions involved sharing with other people, the use of markdown cells and code comments, the size of code cells and notebooks, and version control. Specific questions were generated on the fly in response to interviewees’ answers: e.g., “You mentioned that you’ve used version control, and you’ve also used a sort of numbering scheme. Is there any reason why you do one or the other?” Due to time constraints and which topics a participant expressed the most details on, not all interviews touched on all topics.

Analysis

After transcribing the 21 interviews, I did a line-by-line open coding on a random sample of six transcripts and produced a coding guide. For example, codes for prototyping, experimenting, testing, and iterating were abstracted into Testing Ideas. Following the advice in [Corbin and Strauss 1990] about avoiding bias through allowing scrutiny of the analysis by others, we used inter-rater reliability as an indication that these codes were meaningfully defined. Our collaborator Bonnie E. John used the guide to code one of the transcripts, disagreements were discussed and the definitions improved. Both Bonnie and I coded another transcript with the new guide and attained a Cohen’s Kappa of 0.82. Bonnie E. John then coded all transcripts using the new guide. Codes continued to evolve, primarily in sub-topics, e.g., “Annotating” became “Annotating with Markdown” and “Annotating with Code Comments”.

Limitations of the Data Collection and Analysis Method

Interviewing attendees at JupyterCon2017 is an example of “convenience sampling”, as target participants were congregating at a single place, for a short period of time, and we could obtain permission and physical space from the conference organizers. Interviewing was a condensed and intense activity, with no time in between interviews in which to interweave analysis as is

normal in GTM. Therefore, this study should be considered the first step in a research program and we provide recommendations for theoretical sampling at the end of the chapter. Although four participants showed us examples of their work, subsequent data collection should consider doing field-based contextual inquiry [Beyer & Karen Holtzblatt 1997] to better ground the data in observable behavior.

Results and Discussion

Interview participants will be referred to as N01 to N21 (see Table 1). First we discuss participants' use cases in the notebook environment, and then their iteration behaviors.

Use Cases

Our data revealed three use cases for notebooks: (1) preliminary “scratch pad” work, (2) work that ends up extracted out of the notebook for use in a production pipeline, and (3) work intended to be shared in different ways. These use cases are not disjoint. Pieces of code from a scratch pad can be extracted out into a script for use in a production pipeline. Code extracted for production is sometimes also shared with others via a notebook complete with graphics and detailed explanation.

Scratch Pad Use Case

Almost half our participants (10) used notebooks as scratch pads; 6 explicitly used that term. By this they meant they wrote code cells that they expected to be preliminary and short-lived. Scratch pads were used to answer a specific question, such as how to debug a piece of code, test out example code from the internet, or test if an analysis idea was worth pursuing further.

“I was just testing to make sure I had the syntax right on these tuples.” – N13

“OK so can we do k-means on this dataset and like does it make sense” – N11

Scratchpad use appeared on two levels. “Scratch cells” were used within a notebook during exploration.

“only the last [cell] you did is actually useful so you get rid of some of this sort of trial and error-y things” – N18

At a higher level, some participants had whole notebooks that they used only for scratch work.

“In fact, I have a whole scratch directory where I just run a Jupyter server there and make a notebook, do something real quick and that’s easier than just about anything else.” – N17

Since scratchpad work was intended to be short-lived, participants did not spend time annotating it. However, when the results of scratchwork were successful, that code was extracted out to a permanent place or transitioned from a scratchpad to a substantial literate document.

Production Pipeline Use Case

Seven participants reported that finished code was incorporated into a bigger code base. Finished code had to be extracted out of the notebook and placed in a plain-text code file when

it was needed in a larger production pipeline because the extra metadata in a notebook file made it unusable by automated processes.

Sharing Use Case

Almost all our participants (20 of 21) shared the results of the analysis in a notebook, or the notebook itself, with someone else. These notebooks were typically significant explorations, including developing a model, conducting computational research, or creating a comprehensive analysis. Participants iterated on these analyses over the span of days to months, and generally took care in adding structure to these notebooks, as we will discuss later. Five participants were teachers who gave notebooks to their students for structured assignments:

“We do all of our teaching through notebooks. This includes lectures... in notebooks... then we give them projects to work on... in the form of notebooks.” – N17

Two other participants shared notebooks with clients:

“not only for the actual data exploration myself but then to communicate the results of that effectively back to the person that asked me to do the work.” – No8

Twelve shared with collaborators or teammates:

“I think the reason I use Jupyter is because it actually allows me to share the process by which I arrive at results with people who I want to convince of something, both so that they can spot any errors I may have made and also that they can use similar techniques in their own work.” – N18

Two referred to sharing with their future selves:

“And the idea is to comment them enough... [if you] came back next year would you understand exactly what the notebook was supposed to be doing.” – N13

If any sharing was anticipated, including with a future self, participants reported putting extra care into making sure the notebook was clear to read.

Although notebooks themselves were shared, especially by teachers, many other formats for sharing were mentioned. For example, both financial analysts said they shared results in Excel. Other participants mentioned JPGs, HTML, PDFs, and slides. Today, this is often a necessity:

“most of my clients don't have Jupyter installed on their machines so I can't just give them a notebook file.” – No8

On the other hand, five participants were enthusiastic about sharing “interactive widgets” which can be added as extensions to a cell to allow a reader to tune variables:

“Being able to have them... play with things will be a huge step forward... that would definitely have to be a centrally hosted kind of thing, because we're not going to expect anyone to download Jupyter” – No8

Iteration Behaviors

Organizing the notebook while iterating

Participants reported different strategies for organizing their notebooks while iterating on the code. For example, the bottom of the notebook was used in idiosyncratic ways: two participants reported coding top-to-bottom so the most recent code was always at the bottom; one did all debugging at the bottom and often left it there; one put previously-written functions in a section at the bottom labeled with a “big markdown title” (N11). Two participants put function definitions at the top whereas another used the top to import data. One participant took care to put all cells that loaded external packages in the same place, whereas another participant loaded external packages throughout the notebook.

Another repeated theme about notebook organization was adding new cells directly to where in the notebook the original analysis took place (mentioned by 4 participants):

“if I have to iterate a part of it then obviously I tried to do it close to the place where I inserted it previously, so either in the cell above or below” – N09

This created implicit thematic regions of the notebook where an idea and alternatives to that idea were clustered.

Notebook constraints encourage “expand then reduce” behavior: 8 of the 21 participants explicitly mentioned that they tried to organize their notebooks so each code cell represented a logical unit in the analyses. However, this structure usually came about after cleanup. Participants reported a range of 1-70 lines of code in a single cell, but during active exploration, programmers instead favored creating many small code cells, often only 1-2 lines of code at a time, to incrementally test and build up functionality. This “expand then reduce” pattern was reported by six participants.

“So at the beginning it's usually a lot of little code cells that are one at a time... just making things work... I end up with this huge mess where there are several threads in sort of the same series. So I usually go back and start deleting things or combining cells” – N17

After expanding on an idea, the reduce step is where participants talked about actively “cleaning” up code cells (6 participants) by deleting those they did not need anymore and consolidating working cells into one code cell that represented a logical unit.

Why was the expand step necessary? First, expanding an idea into many small code cells enabled a programmer to pick and choose which cells to run, and thus quickly test different approaches to the same problem. Second, having small code cells allowed a programmer to view cells of intermediary output after each code cell, making it easier to view and reason about their iteration. Third, some participants, although experts in their own domains, were not expert programmers. One participant (N16) referred to the notebooks as a “crutch”, because as a programming novice he felt the notebook had much more support for debugging one line at a time than a standard code editor.

Although generating small code cells was common, it became impractical to leave them all there for the long term. Participants complained that many loose code cells made the notebook a “mess” (a term used by five participants) and more difficult to understand:

“I’ll clean up as I go because otherwise it would be very difficult to be remembering all that stuff” – N5.

The expand-reduce behavior was often talked about in context of fairly low-level exploration, such as building up a working function, or figuring out appropriate library calls. Participants also talked about cleaning up after more significant explorations. For example 11 of 21 participants actively “reduced” their experimentation history by deleting alternatives of an idea from the notebook, or even deleting entire analyses that ultimately proved less fruitful. Although it would be less effort by the programmer to leave prior work in the notebook untouched and only add new work below, instead, programmers took active effort to continually delete scratch work from the notebooks. The attention to cleanup stands in contrast to prior work in non-notebook environments that has reported that programmers have low-investment in tidying code during exploratory data science programming [Brandt et al., 2008].

Notebook constraints encourage managing the length of notebooks: Although a Jupyter notebook does not stop a programmer from adding unlimited content, for pragmatic reasons participants reported that the notebook interface does not work well with long documents. Four participants said that a long notebook was difficult to manage with scrolling up and down. Two others said that when code cells were distantly separated, the code was hard to comprehend.

Because programmers kept different alternative analysis code in the notebook at the same time, they did not want to press the “run all” button to execute all code cells. Instead, participants ran analyses by picking and choosing individual code cells to run. This sometimes required going to the top of the notebook to rerun their standard code cells that import libraries and read in the data, and then scrolling back down to their current work. Scrolling to the top and down repeatedly over a long notebook became a burden.

The practical limit of a notebook, one participant (N16) said, was about 60 code cells. After the notebook got too long or too cluttered, participants would either stop and curate the notebook by deleting alternatives no longer needed or start a new “fresh” notebook, copying in the most successful parts of the old notebook.

“when I open a notebook and I have to scroll for a long time... I just move on to a new notebook” – No3

It is unclear if this is a flaw or a benefit of the notebook design, because the de facto length limit encouraged data scientists to curate which ideas to retain moving forward.

Reuse, reduce, recycle (code)

Almost all participants (19) talked about reusing code. Of those, 11 simply used copy-and-paste. Four reported copying code cells within a notebook to keep code dependencies next to new code. Eight participants copied code into a different notebook:

“I’ll be like, I remember I did this for this project but I can’t remember exactly how to do it. So I’ll go find the project and look at my code and copy paste into the other one.”
– No5

In addition to copy-and-paste to reuse functionality, five participants defined functions and six extracted code into an external script that could be imported into any notebook. For instance, N11 created a new utils.py file for each notebook he worked with, in order to put reusable functions in a special place and reduce the size of the notebook. This practice has been encouraged in some science literature:

“As the code gets longer and more stable, it should be split out into Python modules to keep the notebook short and readable.” [Stevens et al. 2013]

However, this routine practice of extracting notebook code out into a plain Python (or Ruby, Scala, etc.) file for reuse is akin to “throwing the baby out with the bathwater” in that by discarding the notebook’s metadata, the data scientists are also discarding their annotations, graphical output, and richness of exploration that shows how they derived that chunk of analysis code.

Narrative Structure of Notebooks

As in literature, the narrative structure of a notebook that tells the story of the analysis can be linear or nonlinear. A pure linear structure would be akin to paper laboratory notebooks that keep a complete record of every thought, mistake, deadend, and conclusion, in chronological order, often to preserve dates for patent purposes [Tabard et al. 2008]. A non-linear structure could present the story of the analysis as a straightforward progression, recording only important decisions and rationale rather than the circuitous path that actually occurred. This would produce a curated document optimized for comprehensibility over completeness and chronology. A minority of our participants (4) attempted to keep a linear structure, e.g.,

“I have the sort of history of the development upstairs in the notebook.” – No1

On the other hand, most of our participants produced a curated document, e.g.,

“I put the right code where it’s supposed to be and delete the other cells, get rid of it to clean up my code.” – No9

It is important to note that these two goals were contrasting situational goals, and not only individual preferences. Two participants who attempted to create complete records for the purpose of their research also created curated story notebooks. They created curated stories when the goal of that work was to present a specific analysis to an audience, and created detailed research records when research, not communication, was their primary goal.

We now turn our attention to how narrative structures appeared in the notebooks.

Explanation Annotations are Rarely Used in the Exploration Phase of Work

Only six participants spoke about annotating their code during the exploration phase of their work. Of these, three used markdown cells primarily as headers to separate sections of the code. Using markdown only for structural organization, rather than explanation woven throughout

the program, is inconsistent with the definition of literate programming. However, three participants did use markdown during exploration to record their thoughts as they went along.

“I just put the [markdown] on some key changing points of the thought” – N06

“I’ll use markdown cells to put any notes I notice like. ‘OK. Here’s a common way that you make a mistake’” – N17

One person used code comments (not markdown cells) as

“a way for me to track my process of going along and to keep thinking through the problem... comments help me think.” – N16

In contrast, after the process of exploratory programming was done, if a participant had a long-term purpose for their notebook such as sharing it or keeping it for a record, they would then add more explanatory documentation to the notebook that is more consistent with literate programming. Nine participants reported this behavior:

“If it’s a notebook that... has to be rerun by me or by somebody else, I’ll try to explain the data sources, where it comes from... And just the different major steps in the analysis,” – N05

“[When] I’m doing research, it’s almost like a source code. And then when I really want to clean it up and show it to someone else, then I put in annotation.” – N10

Mechanisms used to provide narrative structure: The interweaving of input code and output is a primary mechanism of narrative structure of any notebook.

“it’s nice because all of the images are right there and all the code is right there.” – N02

In addition, participants talked about how they used the code itself to provide narrative structure, through which code cells they chose to keep and delete.

“if you read my notebook from top to bottom you see the evolution of my thought. You see that I first do some... small part of the function, then... the universal functions... And finally... the conclusions that are made using the functions.” – N06

During the dissemination phase, participants used markdown to create a narrative structure:

“If I’m putting together a script notebook for someone else to use [I’m] making it nicer and adding markdown and everything.” – N03

Sometimes markdown was used to tell a linear story:

“Not only do I just say [in markdown] what I... removed, but sometimes I show those intermediate steps so that they can see the progression from raw uncleaned data to the final product.” – N08

Other times, markdown was used to curate the story in concert with deleting less important code to make the key points of the exploration more apparent:

“I end up with a really messy notebook and I might end up... opening another one and just doing the clean version... The stuff that worked. And just with more comments and just you know, nicely formatted.” – No5

One participant felt that the narrative structure emerged as he cleaned up his code:

“I usually go back and start deleting things or combining cells or shifting things around... so the eventual form with the notebook only gradually emerges” – N17

In contrast, participants who used a linear narrative structure made earlier cells in their notebook historical and immutable by avoiding overwriting code cells that had already produced output, and added new code cells only to the bottom of the notebook. This meant a series of code cells that perform a data transformation might be duplicated at different locations in the notebook, enabling the author to keep different output for each variation and retain a chronological record. This completeness came at the cost of a hard to read narrative:

“I can't get an overview of what's going on in my notebook... it's just a lot of stuff and stuff and stuff... with all these random outputs that never get cleaned up.” – No1

Although these participants achieved a more detailed record by avoiding curation, it should be noted that they necessarily curated each time they decided whether to overwrite and re-run a cell or create a new cell.

Version Control

The vast majority of participants spoke about iterating extensively on their code (only 2 of 21 said their code development progressed in a straightforward fashion). However, all this exploration was generally thrown away. Participants identified why current means of versioning with literate programming notebooks is fairly dysfunctional. Recall that improved version control was the most requested feature in the Jupyter Project 2015 UX Survey [Jupyter Project UX Survey 2015].

While 11 of our 21 interview participants did use a version control tool like Git for their notebooks, the metadata included in the file format of notebooks currently makes Git utilities such as diff (viewing the differences between two source code versions) unusable because utilities were not designed to treat metadata differently from source code.

“diffing is so hard...I develop until I'm happy and then I'm going to put it in a file and then I'm going to version control the file not the notebook.” – N15

Although a technical annoyance, two participants' workplaces had scripts to extract the code out of a notebook and just version that, enabling a normal Git workflow.

Some participants appreciated the conditions under which formal version control like Git or SVN is important:

“If it's an application, usually there's all sorts of dependencies. And that's when version control becomes important. Also if... I have to release this in concert with something else... then you have to do some sort of version control” – N12

Because of the effort involved in using formal versioning tools, participants often used informal versioning. Consistent with our prior observations of informal versioning practices (Chapter 3), 4 of 21 participants relied on different file names for version control:

“The stupidest possible version control... you rename the notebook to something like V0 or V3.” – N18

This informal method has its own problems:

“...we have like 500 different files all variations of the same thing and they're all numbered in a way that's completely useless because I don't remember whether it was two weeks ago or two months ago I was at this stage of the iteration.” – N12

Also consistent with Chapter 3, participants used local versioning inside their notebook. For instance, two participants said they had code cells containing alternative approaches simultaneously in view to be able to compare them. However placing alternative code and output cells directly above or below the original was a problem due to screen space. With large code or output cells, authors could not see everything they needed at once in a single notebook window. Two participants reported workarounds in order to see alternatives side by side, for example opening two different windows of the same notebook to place the windows side by side on their screen.

QUERY DESIGN EXERCISE

We drew inspiration from the “grounded brainstorming” procedure described in [Beyer & Holtzblatt 1997] to design a short computer survey to elicit data scientists’ versioning needs. The survey first grounded them in their real experience by asking them to describe a recent exploratory data analysis they had performed (Q1). The survey then primed them to think of an imagined future with a *“magical perfect record of every analysis run you did in this project. You also have a magic search engine that can retrieve for you any code version, parameters used or output from the past.”* After this preparation, we asked people to brainstorm by typing *“as many queries as you can think of that could be helpful to you to retrieve a past experiment. Don't worry about feasibility.”* We asked participants to “phrase [your query] in natural human language like you're talking to a colleague” both to discourage the participant from assessing feasibility and to provide phrases we were likely to understand (Q2). Finally, we probed for the types of real world problems such future magic technology might be able to solve: *“Has **not** being able to find a past experiment ever caused you problems? If yes, what happened?”* (Q3).

The survey was conducted at JupyterCon 2017 on a laptop (27 participants) and online (18 participants), advertised through posts on the first author’s social media inviting data scientists to participate.

Analysis

Treating the participants’ answers to Q2 as brainstorming ideas, we used affinity diagramming to cluster the imagined queries into different categories (Table 4.2). We performed a separate affinity diagramming to cluster participants reported problems (Q3) into categories.

Table 4.2. Affinity diagramming groups for 125 queries. A query can appear in multiple groups.

Referenced analysis attribute	# Queries
Analysis (e.g., “convolutional model”)	46
Output (e.g., “training accuracy”)	25
Time period (e.g., “go back 5 hours”)	17
Dataset (e.g., “previous test result for this particular dataset”)	15
Plot (e.g., “how did I generate plot 5”)	11
Specific variable	10
Library	4
Running time of the program (e.g., “How long did it take to process country X”)	3

Results

All survey participants will be referred to as SP01 to SP45. In Q2, 45 participants generated a total of 125 queries¹² for the “magic search engine”. Participants’ queries referred to many kinds of contextual details, including libraries used, output, plots, data sources, parameters used, running time of an analysis, time periods, version numbers, and specific dates (Table 4.2). Participants did not limit themselves to imagining only prose queries, e.g., SP13 submitted “Here’s a visualization I produced, let me right click on it to give me the script to produce it”.

In addition, some queries required semantic or conceptual understanding of the programmer’s task, for instance “Show me all the different ways I oversampled the minority class” (SP21), or “What was the state of my notebook the last time that my plot had a gaussian-ish peak?” (SP17). Some participants also asked for properties of an analysis relating to process or rationale, for example: “Find me how I cleaned the data from start to finish” (SP08) or “What questions did I ask that didn’t pan out?” (SP12).

For Q3, 31 of the 41 participants who answered said they had experienced problems being unable to find prior analyses versus 10 who had not. The most-mentioned problem was the need to rewrite code (20 participants). This need had several sources, including losing the code because that part of the work had not been saved or losing the rationale behind the code because it had not been recorded. Without the code that produced a result, 7 participants no longer trusted that result. The second most frequently reported problem was time delays (12 participants) caused by excessive time searching for code, having to re-run code, or having to rewrite code. Two participants reported having to consult with a colleague to solve the problem.

The answers to Q3 validate prior findings [Klokmos & Zander 2008, Tabard et al. 2008] that past analyses can be hard and sometimes impossible for data scientists to find. In notebooks,

¹² A full list of queries generated by participants can be found in Appendix B

version control is currently poor enough that records of prior iterations often do not exist. Yet even with improved version control, it should be noted that some ‘magic’ queries from Q2 cannot easily be translated into traditional text search-engine queries, e.g., “the last time that my plot had a gaussian-ish peak”.

LIMITATIONS

Our data analysis has produced accounts of three types of use cases, a variety of mechanisms for narrative structure and version control, and several design directions. As mentioned, these results can be viewed as hypotheses in a longer research program and future studies should consider using theoretical sampling in the following three ways.

First, our convenience sampling did not screen for profession or domain of study, and our data suggest that different professions do different things. For example, the financial analysts used Excel for dissemination; the computational biologist said clients do not have Jupyter; and teachers shared notebooks, but tended not to create production code. Future research should sample from different professions, especially if designers want to produce tools for specific user groups.

Second, we studied only Jupyter Notebooks and the iteration behaviors we observed may have been influenced by specific UI details. Future studies may want to sample other tools to find which behaviors generalize.

Finally, perhaps most importantly, the data about participants’ behaviors was self-reported, not observed, so future studies should seek to verify these hypotheses through direct observation e.g.,

[Beyer & Holtzblatt 1997] or fine-grained logging that could confirm behaviors like expand/reduce.

CHAPTER CONCLUSIONS

By talking with practitioners directly, in this chapter we learned more about the iterative process behind developing notebooks. Here I’ll summarize some of the key design ideas that this chapters’ studies contribute to our history tool goals of this dissertation.

Importantly, we learned not just about the notebooks people author *for sharing* (of which there are many examples shared on the web), but also messy scratchpad notebooks for quick experiments, and those notebooks intended for later integration with a production pipeline. These use cases likely have different needs for history support:

- **In the case of the scratchpad**, just like its physical pen-and-paper counterpart, much of what is done in a scratchpad notebook is assumed to be temporary and disposable. Participants did not express any need for history support for such a notebook. However, for scratch cells in a more substantial work notebook, or for scratchpads that evolve into more substantial work notebooks, history may be helpful to demonstrate where more permanent work came from.
- **In the case of notebooks that develop model or analysis components intended for use in a production pipeline**, the notebook itself won’t be used in production. Instead, we might think of history in terms of provenance support: keeping ties between a model in

production and the notebooks that demonstrate how that model was created, tested, or debugged.

- **In the case of notebooks intended for sharing,** we need to consider the history of a notebook as something that will be shared too. How might history support *someone else* in understanding what the original author tried? How might history be summarized or displayed in such a way to support someone else gaining an *overview* of what work was done?

The iterative behaviors we found in notebook development also impact how history support should be designed:

- **History can provide a safety net for organizing the notebook.** Since practitioners often curate as they work, removing less successful data work, a clear value proposition for history support is that we will be able to *preserve* these discarded parts of work while letting users keep a tidy notebook that contains just their most recent work. The small number of participants we interview who “keep everything” in their notebooks may feel more flexibility to organize and curate their document if they don’t have to worry about losing content.
- **“Expand then reduce” cell iteration means that cell-level versioning will not be sufficient.** Since practitioners develop code across multiple cells before combining it into a single cell, the history of a given code chunk will have provenance in multiple cells. As compared to seeing the history of a single cell, this is a more complex history structure that we need to support users tracing and understanding.
- **Most notebooks have a narrative cell structure top to bottom.** This means that a historical cell’s *location* in the notebook is likely to carry some semantic information about what the cell is about.
- **Explanatory notes are minimal during exploration.** Since many participants do not take the time to add markdown notes or comments at the stage of active development work, we should not *depend* on users to take the time out during active development to annotate their history. Just like some users use comments and markdown to mark important ideas in their thoughts, users may want to label a version of their history if it is particularly important..., but we should not *rely* on having semantic version labels provided by users for most versions.

Finally, the Query Design Exercise provides us with hints on how practitioners recall versions that are important to them, and how they might go about finding those versions. These different categories of memory queues are shown in Table 4.2, and each will likely need a different tooling approach. For instance, runtime information will need to be originally recorded in some way for it to be searchable in history. For visual queries about plots, either some machine vision processing will be needed to make plots searchable by text or we will need to design a UI such that it is easy for a user to manually skim through the historical plots to look for a certain attribute.

Overall, our study of notebooks demonstrates that notebooks are indeed a powerful tool for organizing and structuring exploratory programming data work – it is just that notebooks and narrative structure alone are not sufficient forms of history-keeping to meet practitioners’ needs. Participants used notebooks to keep “*the sort of history of the development upstairs in the*

notebook.” but notebooks as history were not particularly usable: “it’s just a lot of stuff and stuff”. This is an important lesson for development of history tool support. In taking on the role of holding the “*history of development*,” how will a history tool avoid similarly becoming just a pile of “stuff and stuff”? **Supporting search and comprehension of history will be just as important as effectively storing history for users.**

Part II: Designing Better Support for Experiment History

INTRODUCTION TO PART II

In Part II of this dissertation, we prototype a series of different approaches to history support for exploratory programming data work. First in Chapters 5 we survey history support tools and techniques in prior work, and then in Chapter 6 describe our overall design process. Chapters 7-12 detail a series of 6 prototype history systems, all of which sit directly in a data worker's active editor as they work. Through each iteration, we illuminate new characteristics of the design space and push our understanding further around what practitioners need for effective history support. The final series of 4 prototypes develop a tool, Verdant, which we release as a history extension for the JupyterLab computational notebook environment. This final deployed version of Verdant we then test in Part III of the dissertation.

Chapter 5: Related Work on History Systems

In software engineering, version control (also known as revision control, source control, or source code management) is a class of systems responsible for managing changes to computer programs, documents, large web sites, or other collections of information.

– Wikipedia: Version control, retrieved June 2020

INTRODUCTION

In this chapter we cover a variety of system approaches to history and version control. We first cover fundamental models of history and interaction in popular source code management tools like Git. Next we cover what history support (currently) exists for notebooks. Finally we cover alternative visualizations and models of history from HCI research systems. Many of the systems covered in this chapter directly inspire UI or technical design patterns that we adopt in our own history systems in this dissertation.

GIT & SOFTWARE ENGINEERING VERSION CONTROL

In this research we heavily draw inspiration from standard version control systems (VCS) to adapt their benefits into exploratory programming. In contemporary software development, VCSs such as Git, SVN, or Mercurial are so ubiquitous that ignoring their pre-established conventions in our design would be ill advised. We may improve adoption by developers if our designs fit with, rather than conflict with, the common Git/Mercurial/SVN version control tools they are used to. We can also avoid redundant effort for technical issues not specific to exploratory programming that standard VCSs already solve. Git is the most popular VCS in the world for code (at least among open-source repositories since 2018¹), and is our primary inspiration for how versioning works in all VCS prototypes of this dissertation. To help the reader better situate design choices we make in our own VCS designs for exploratory programming with data, we briefly cover some of the conventions we adopt or adapt from Git:

- **Change/revision/edit/modification:** are all terms used interchangeably to describe what an author changed in a specific document to create a new version.
- **Commit:** we call a specific version of the project a “commit” and use the two terms interchangeably. A commit is a *collection of specific changes* across all documents in the project.
- **Branch:** a sequence of commits make up a branch.

¹ The popularity of different version control systems is hard to measure. People have used metrics like google search ranking, count of questions on stack overflow, large public developer surveys, and counts of publicly available software projects to estimate popularity:

<https://softwareengineering.stackexchange.com/questions/136079/are-there-any-statistics-that-show-the-popularity-of-git-versus-svn>

- **Di.** : a visualization that shows the text edit differences between two versions (commits). Some special diff techniques can show the perceptual visual differences between two images, but a diff in Git is typically plaintext only.

Below in Figure 5.1 is a typical Git history graph. A history graph, or “revision tree”, visualizes all of the branches and commits of a project. The numbers at the top of the graph denote dates, from the end of June to July 23rd. Each bolded color line on the graph represents a different branch of the project, with the black branch being the **master** branch. Each dot along the branch lines is a commit. The arrows denote when branches were **merged** to combine their parallel histories. Although it may look chaotic, this graph shows a typical project workflow. Focussing on the end of June and July 1st, the following actions happened:

- (Prior to June 29th) Alice is working on a new software feature so she began a new branch (**blue**)
- (Prior to June 29th) Bob also started a major update and began a new branch (**purple**)
- (June 30th) Bob merges a fix from his branch back to master (**purple → black**)
- (July 1st) Bob merges a fix from his branch back to master (**purple → black**)
- (July 1st) Terry also starts a new feature and so begins a new branch (**green**)
- (July 1st) Alice pulls Bob’s fix from master to use with her feature (**black → blue**)
- (July 1st) Alice merges her changes into the master branch (**blue → black**)
- (July 1st) Terry pulls Alice’s updates to the master into their branch (**black → green**)
- (July 1st) Bob pulls from Terry’s branch to continue work on his branch (**green → purple**)

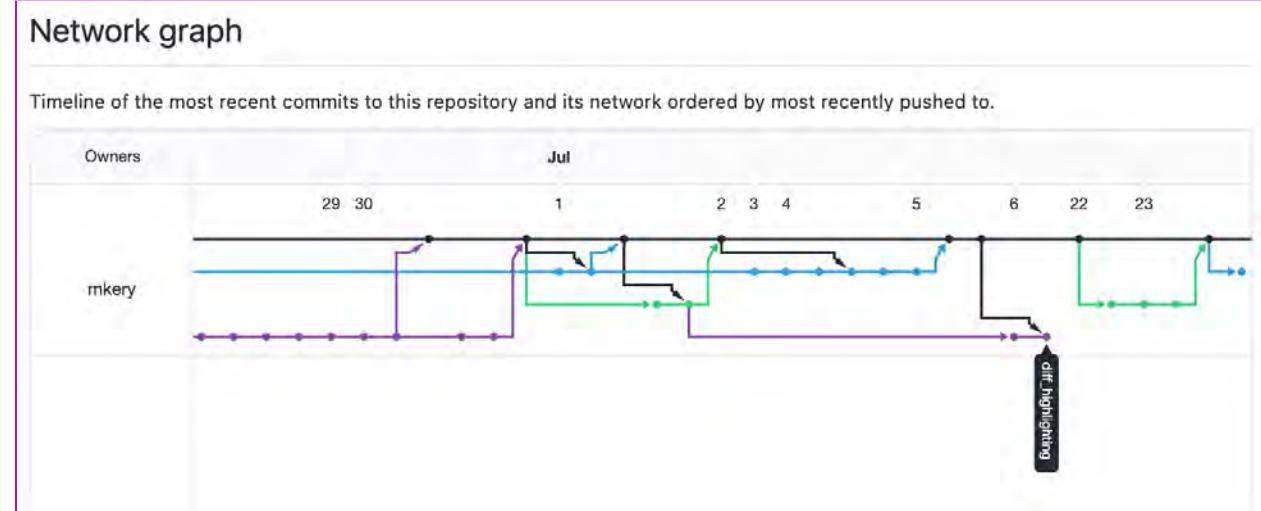


Figure 5.1 GitHub’s visualization of commits and branches of a project during a period from June to July. Github is the most popular platform today for hosting Git projects.

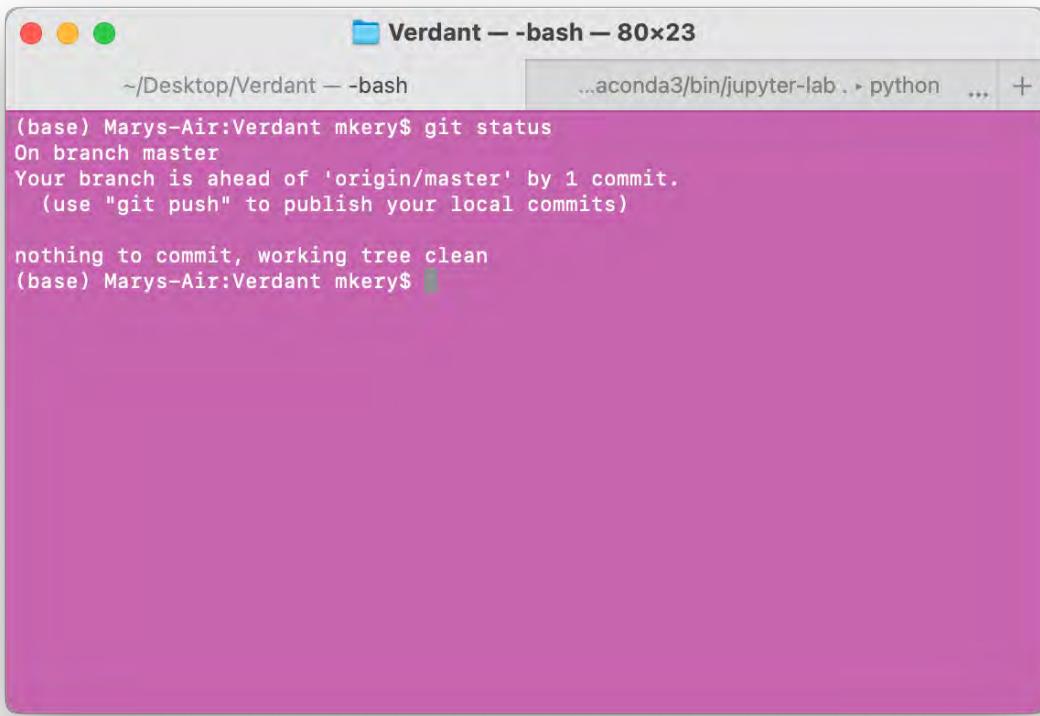
Different branches can generally be thought of as side-projects or sub-tasks of the overall software development process that are developed outside of the master branch and then eventually merged with it. Branches allow developers to experiment with new ideas that might temporarily introduce new buggy behavior while ensuring the main code on the master branch

remains safely stable and untouched. In this dissertation we scope our efforts and do not deal with branching nor with collaboration due to the complexities those introduce.

User Interfaces for Git

Here we cover the three major settings where a user will interact with their Git history, each of which has its own ecosystem of user interaction conventions.

First and foremost, a user interacts with Git through a command line interface (CLI) in their terminal shell. The terminal CLI (provided by the Git program itself) remains the dominant setting where most programmers first learn Git and is almost always what is taught in any educational material on Git. Figure 5.2 below shows a typical Git interaction, which is purely textual. CLI commands are extremely powerful for Git, but many programmers know only the most basic everyday Git commands². The CLI does not provide much flexibility to visualize alternative versions or visualize history at all, as we discussed as a user need for exploratory programming in Chapter 3. It is worth noting that in addition to Git, other common version control systems like SVN or Mercurial are all CLI based.



```
Verdant — -bash — 80x23
~/Desktop/Verdant — -bash ...aconda3/bin/jupyter-lab . ▶ python ...
(base) Marys-Air:Verdant mkery$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
(base) Marys-Air:Verdant mkery$
```

Figure 5.2 A terminal showing a git CLI command `git status` and its output

Next, for those programmers that use integrated development environments (IDEs), most IDEs have built-in tooling for version control. Below, Figure 5.3 depicts a variety of version control features available in the popular IDE Visual Studio Code by Microsoft. Within the editor, these

² There is no data I am aware of that depicts how much an average programmer knows about Git or version control, but it is a matter of “common knowledge” that Git is hard to learn and understand, as evidenced by this popular xkcd comic <https://xkcd.com/1597/>

version control interactions are primarily about diffing to compare the current code a person is working on to some other version or else about choosing which branch or version to work on in the editor. Many of these interaction features we can partially imitate in an exploratory programming use case, but not exactly copy. The chief difference in our exploratory use case is that the user needs to interact and work with *many versions of something at the same time* in their working environment, whereas in the Git workflow, a person has only one version in their working environment at a time. Like the command line interface, note that these code editor UI features are not exclusive to Git. For instance in Fig 5.3, these interactions appear almost the same in VS Code regardless if the user is working with Git, SVN, or Mercurial.

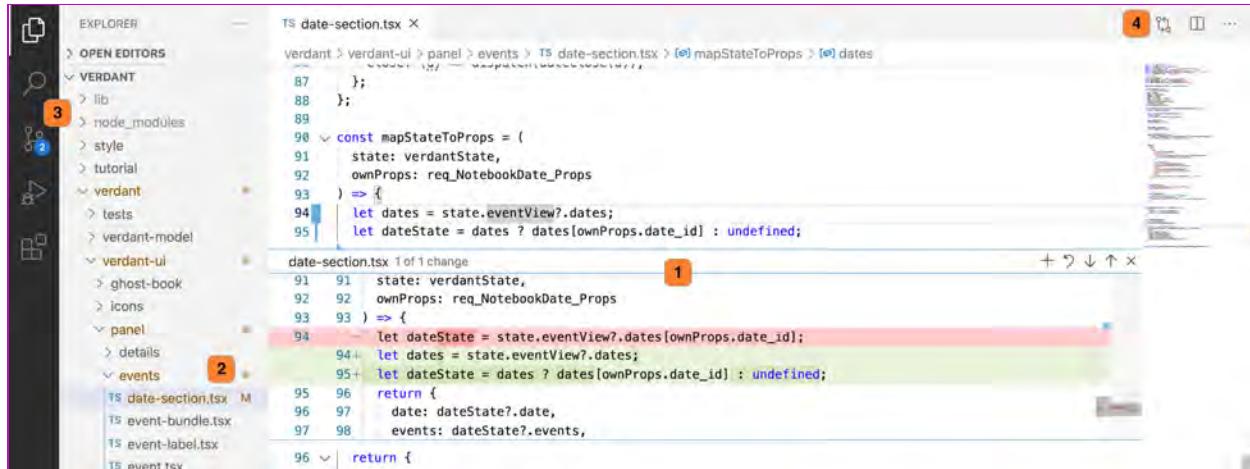


Figure 5.3 In-editor Git interaction types in Visual Studio Code³ in 2021. At (1) if the user clicks a section of their code that they have changed, but not yet committed to Git, they can trigger a pop-up in-line view that shows a diff of how that section has changed from the last Git version. At (2) ambient indicators show brown-colored folders and dots to indicate that something in that folder has been changed since the last Git version, with a brown “M” next to the file `date-selection.tsx` to indicate the file was modified. At (3) the user can click the versioning icon in the navigation bar to see a list view of all files that have been changed since the last version. At (4) the user can click on the versioning icon to see a full side-by-side diff view of their currently open file versus the prior version.

Finally, the web browser is a critical location for interacting with the more project management side of software version control (Figure 5.4). Providers such as GitHub, GitLab, or Bitbucket provide graphical tools for the entire team to view and collaborate on projects. The purpose of most of this graphical tooling is for navigating a large collaborative project and project management. Open source projects often live on GitHub as public projects where anyone on earth can join the project team as a contributor and where anyone can look up any other public project. This means that the remote copy of the project is stored on GitHub’s servers. Open source projects are a critical core part of our contemporary technical infrastructure for the entire world. However private projects or companies with their own internal servers to store

³ Visual Studio Code <https://code.visualstudio.com/>

their own remote projects still often use GitHub's graphical user interface because of all the valuable project management features that come with the web interface for a Git project. Again, comparable tools exist for the other version control systems like Mercurial or SVN.

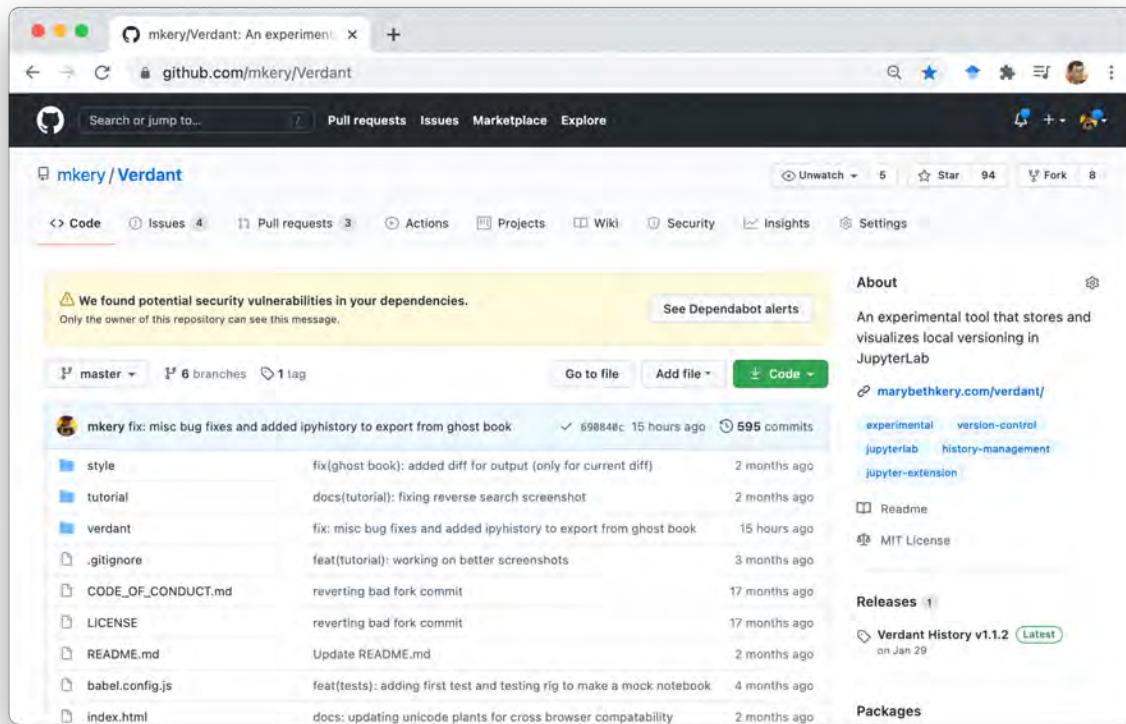


Figure 5.4 The GitHub main page for a project, here the Verdant tool project from this thesis. The left side of the screen shows all folders and files in the project, as well as their latest commit message and a time indicator of when they were last updated.

There are graphical user interface equivalents to the CLI tools, for example GitHub Desktop, which offer a small subset of the CLI's commands but also a subset of GitHub's rich diff visualization tools found on the web platform.

Universal Diff Notation

To conclude our primer on version control conventions, a word on diff notation. Showing the difference between two documents (diffing) has style conventions that are universal to versioning tools, and thus appear in our own system designs (see Figure 5.5). First, the color green and a '+' plus sign always denote text that was added. The color red and the minus sign '-' denote text that was removed. The color blue does not typically show up in the diff itself, but is commonly used in version control UI, sometimes with the symbol 'M' as in Figure 5.3, to denote a change or modification in general.

```

69  let matchCount = 0;
70  notebook_history.cells.forEach(name: string) => {
71    let nodey = this.history.store.get(name);
72
73 -   let match = notebook_view.widgets.findIndex(w => {
74 -     if (w instanceof MarkdownCell && nodey instanceof
75 -       NodeyMarkdown)
76 -       return nodey.markdown === w.model.value.text;
77 -     if (w instanceof RawCell && nodey instanceof NodeyRawCell)
78 -       return nodey.literal === w.editor.model.value.text;
79 -     if (w instanceof CodeCell && nodey instanceof
80
81  let matchCount = 0;
82  notebook_history.cells.forEach(name: string) => {
83    let nodey = this.history.store.get(name);
84 +   let oldText = this.getText(nodey);
85
86 +   let match = notebook_view.widgets.findIndex((cell, index) =>
87 +     {
88 +       let newText = this.getText(cell);
89 +       return oldText === newText && !toMatch[index];
90

```

Figure 5.5. A screenshot of GitHub’s split diff view of a commit

```

@@ -91,7 +91,8 @@
 const mapStateToProps = (
  state: verdantState,
  ownProps: req_NotebookDate_Props
) => {
- let dateState = state.eventView?.dates[ownProps.date_id];
+ let dates = state.eventView?.dates;
+ let dateState = dates ? dates[ownProps.date_id] : undefined;
  return {
    date: dateState?.date,
    events: dateState?.events,

```

Figure 5.6. A screenshot of a diff in the Git command line interface

There are two standard ways to show a diff. A “split” or “side-by-side” diff view as in Figure 5.5 puts each version of the document to one side. Conventionally in a split view, the older version of the document will show all deleted changes and the new version of the document will show all added changes. In an “inline” or “unified” view as in Figure 5.6, added and deleted changes are shown together.

VERSION CONTROL FOR NOTEBOOKS

Git used as-is with notebook documents works poorly due to the amount of metadata and output data in a notebook file. Although many functions of Git still work as intended, differencing or merging two notebooks fails. When Git attempts to do its line-level diff on output or metadata that cannot and should not be split into lines, Git ends up creating a nonsensical and broken jumble of text. Since our research began in 2016 several tools have emerged to handle this issue. Project Jupyter added the nbdime⁴ tool to show a web-based diff of notebooks. A commercial startup ReviewNB⁵ offers similar differencing to nbdime as well as tools for commenting and code review on notebooks. Other notebook platforms like Google Collab (see Figure 5.7) have their own built-in diff viewer for notebooks. All of these tools offer approximately the same thing: a custom app on top of Git to make comparison work correctly for notebooks.

⁴ nbdime by Project Jupyter <https://nbdime.readthedocs.io/en/latest/>

⁵ ReviewNB <https://www.reviewnb.com/>

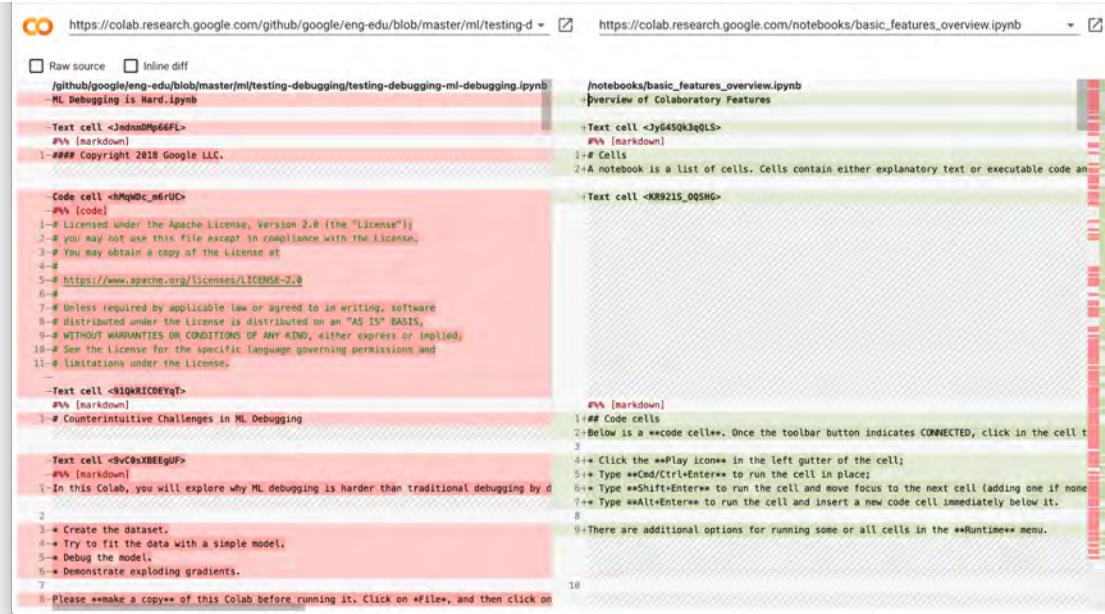


Figure 5.7 Notebook comparison tool in Google Colab

Our own research overlaps, but has a different focus than “Git for notebooks”. While borrowing many conventions from Git, our goal is to see if we can break history down into smaller, more actionable parts to be directly used during experimentation.

A different side of history is the *run* history. A run history tells a person what code they have run in what order in a given session. In tools like R Studio⁶ a data scientist can see a list of code they have run so far. In a Jupyter notebook the user can run the `%history` command to see the same thing. This kind of history is kept in the tool just for a single session.

OTHER RESEARCH SYSTEMS FOR HISTORY

Dealing with alternatives

Several other research tools have explored interactions for alternatives of code. Juxtapose (Figure 5.8) [Hartmann et al., 2008] is a research tool that provides interaction designers with different alternatives of their code, in order to compare among different parameters of the look and feel of their interface designs. This tool used Linked Editing, a technique for editing two alternative pieces of code simultaneously, previously developed by Toomim et al. [Toomim et al., 2004]. Juxtapose also built off of prior work such as Set Based Interactions [Terry et al., 2004] and Subjunctive Interfaces [Lunzer & Hornbæk 2008], which explored general techniques for exploring multiple alternatives in parallel. These were not specific to writing programs. Originally, we were heavily inspired by Juxtapose and Set Based Interactions in the design of our UI for Variolite (Chapter 7), for instance by the tab-based interface design in Juxtapose.

⁶ R Studio <https://www.rstudio.com/>

However ultimately we found that the specific designs of these systems work best with just a few alternatives. For instance, the tab interface and side-by-side view of Juxtapose in Figure 5.8 show just 2 alternatives. Through experience with our own designs, we found that for history purposes, the user may be easily considering a dozen alternatives (e.g., alternatives of a visualization), which cannot be so easily displayed in side-by-side or tabs layouts.

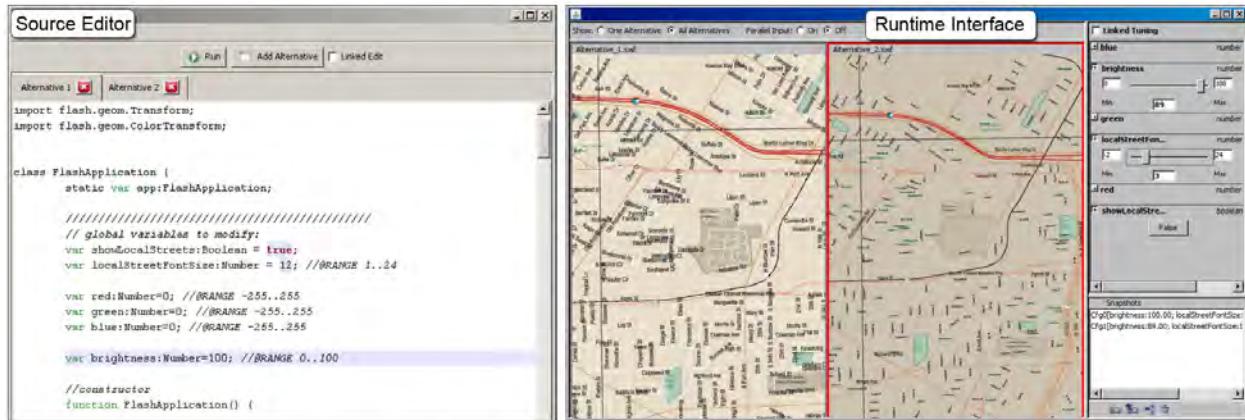


Figure 5.8 Juxtapose [Hartmann et al., 2008]

On the side of professional software engineering, *software product lines* are a method used in industry to adapt one piece of software to be customizable for different clients or devices [Clements & Northrop 2001]. Software product line research aims to handle much more complex versions and interdependencies across an entire software project, with commensurate complexities in the developer's interface. Ultimately this research informs our versioning work as a kind of cautionary tale. Dealing with alternatives, which delightful in just a few alternatives in systems like Juxtapose [Hartmann et al., 2008] or Parallel Pies [Terry et al., 2004], quickly devolves into confusion at scale.

Within a code editor, Yoon and Myers created Azurite [Yoon & Myers 2015] for selectively undoing past actions in code using a timeline visualization. Other interactions for versioning have been developed for End-User Programmers, such as for Mashups where Kuttal et al. showed that versioning helped programmers work more efficiently in Yahoo! Pipes visual programming tool [Kuttal et al., 2011]. All of these history systems use a timeline visualization. While a timeline is a classic and easily recognizable visualization for history, we initially tried but later abandoned the use of timelines in our own designs. This is because in our form of history support, we are more focused on representing how multiple specific artifacts (e.g., plots and models) are changed over time, and a timeline abstracts away most information about content. Later in our design of an activity view (Chapter 11) we create an activity stream visualization, which serves the same purpose as a timeline, but shows more descriptive text and visualizations of content changed as opposed to the simple red and green block visualizations in Yoon's edit timeline.

Interactive History for Data Experimentation

Specifically for machine learning models, ModelTracker [Amershi et al., 2015] visualized a timeline of data scientists' iteration over time (Figure 5.9). Similarly, Patel [Patel 2013] created a research tool called Hindsight, which keeps a history of different parameters used in a

programming task for machine learning classification. Hindsight also allows users to combine different alternatives of steps in the classification, such as which data is loaded and which algorithm is used (Figure 5.10). Although ModelTracker and Hindsight serve as inspiration for our own systems, our own visualization design choices look very different since we are aiming to support any exploratory programming tasks. Both ModelTracker and Hindsight are task-specific interfaces, and have more features specifically tailored to machine learning modeling tasks.

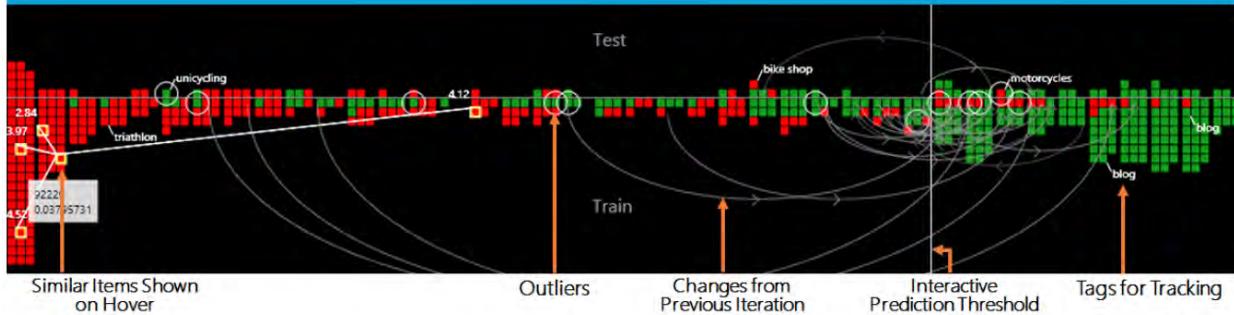


Figure 5.9: Model Tracker [Amershi et al., 2015]

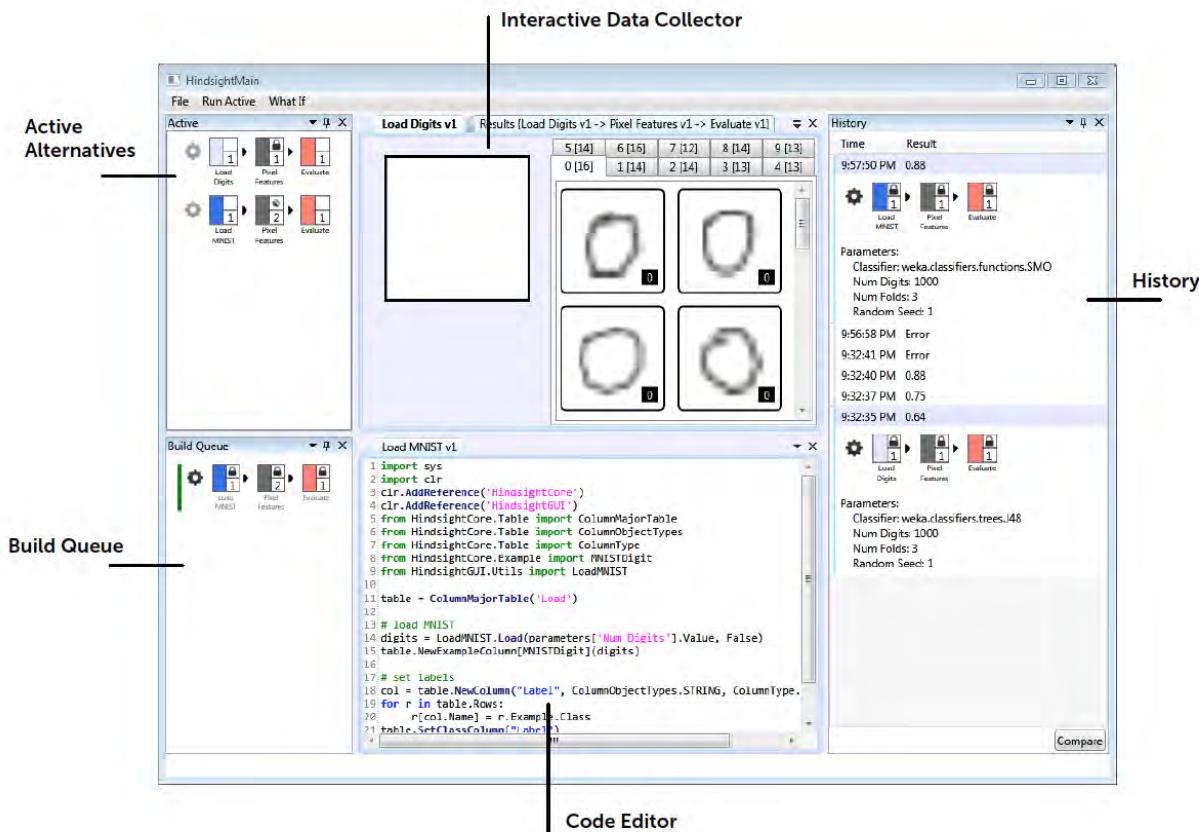


Figure 5.10 Hindsight [Patel 2013]



Figure 2: The Activity Feed resides on the desktop background and shows a near real-time stream of user actions

Figure 5.11 Burrito activity feed [Guo & Seltzer 2012]

Guo and Seltzer [Guo & Seltzer 2012] produced a research system called Burrito (Figure 5.11), which displays a GUI activity feed of things like outputs, save events, and notes relevant to a given project. The Burrito tool collects much more detailed provenance information by working at the operating system level in Linux. The major benefit of Burrito's approach is that it is able to listen for experiment activity across different tools in a user's workflow, which can be very difficult to do since separate software exposes different information about what a user is working on. By capturing information across multiple tools, Burrito is able to capture a full picture of a realistic data workflow. The major drawback to Burrito's approach is that it works at the operating system level. Access to the operating system level is too invasive to be viable in most real usage situations, due to security and privacy issues. The way that Burrito “listens” to a user's computer activity, while very helpful for tracking experiment history, is most similar to

spyware or malware. Thus it remains an open challenge how to make interoperable data workflows across multiple tools in a way that preserves security and privacy.

APPROACHES FOR HISTORY AS CURATION

So far, all of the systems and approaches we've discussed focus around *versions*: some notion of checkpointed *copies* of the user's work over time. However, that isn't the only way to conceptualize history. There are many formats for summarizing or storytelling what work was done. For instance Datasheets [Gebru et al. 2018] is a report format that describes the key properties of how a dataset was derived and how it should be used. Similarly Model Cards [Mitchell et al. 2019] promote a similar short-form report that describes how a model was developed and can be used. As we saw in Chapter 4, computational notebooks like Jupyter notebooks are often used to create computational narratives [Rule et al. 2018], which is an interactive report that walks a reader through how a model or analysis is derived. A key attribute of all of these report-style forms of history is that they are typically reporting on the final outcome of experimentation and exploration, and typically omit all the other ideas explored that were less successful or did not contribute to the final outcome. A report has a level of curation that makes it a more semantically rich form of history than a pile of unlabeled version copies, however at the cost of being selective about what experiments are included. On the other end of the spectrum, a classic lab notebook from wet lab sciences like chemistry or biology is a report format that documents *every* experiment done. The high level of careful documentation in a lab notebook is due to principles of scientific rigour and pragmatic legal reasons⁷. In practice (Chapter 3 & 4) we have not observed this level of note taking or motivation to maintain this level of note taking by practitioners doing programming experimentation.

CHAPTER CONCLUSIONS

Of the systems discussed in this chapter, none addresses the same set of issues that we aim to address in our history tooling design. Classic tools like Git provide extremely helpful technical and UI design patterns, but are missing the ability to work with many, fast paced versions of small content within files at once (Exploratory Programming Study Chapter 2). Research tools like Hindsight or Juxtapose demonstrate compelling visions for letting users switch between alternatives of their work, but lack design patterns for scaling to dozens or hundreds of versions. Finally, tools like Modeltracker or Burrito show exciting visualizations for showing analysis work over time, but again, are not about quick access of versions of specific small content. In our approach to history tooling to support experimentation, we aim to allow users to quickly access specific alternatives of small snippets of their work, but also scale that access to dozens or hundreds of versions that may occur over longer periods of work.

⁷ See the United States National Institute of Health's training on lab notebooks for government-funded research "Keeping a Lab Notebook"
[https://www.training.nih.gov/assets/Lab_Notebook_508_\(new\).pdf](https://www.training.nih.gov/assets/Lab_Notebook_508_(new).pdf)

Chapter 6. Design Process Overview

INTRODUCTION

Moving into the system design portion of this dissertation, our storyline maps well onto the classic “Double Diamond” design process model, moving from the first diamond into the second (Figure 6.1).

In Part I we began by broadly investigating exploratory programming as a phenomenon, before narrowing our research efforts to focus on data scientists as a particularly critical group today who do exploratory programming. From there we converged on a clear need and opportunity for HCI system research: data scientists need (and lack) good history support for doing exploratory programming data. This need for better version control was validated and elaborated on in our research of data scientists working in a variety of code environments (Exploratory Programming Study, Chapter 3), in notebook programming environments (Notebook Usage Study, Chapter 4), as well as prior research by others.

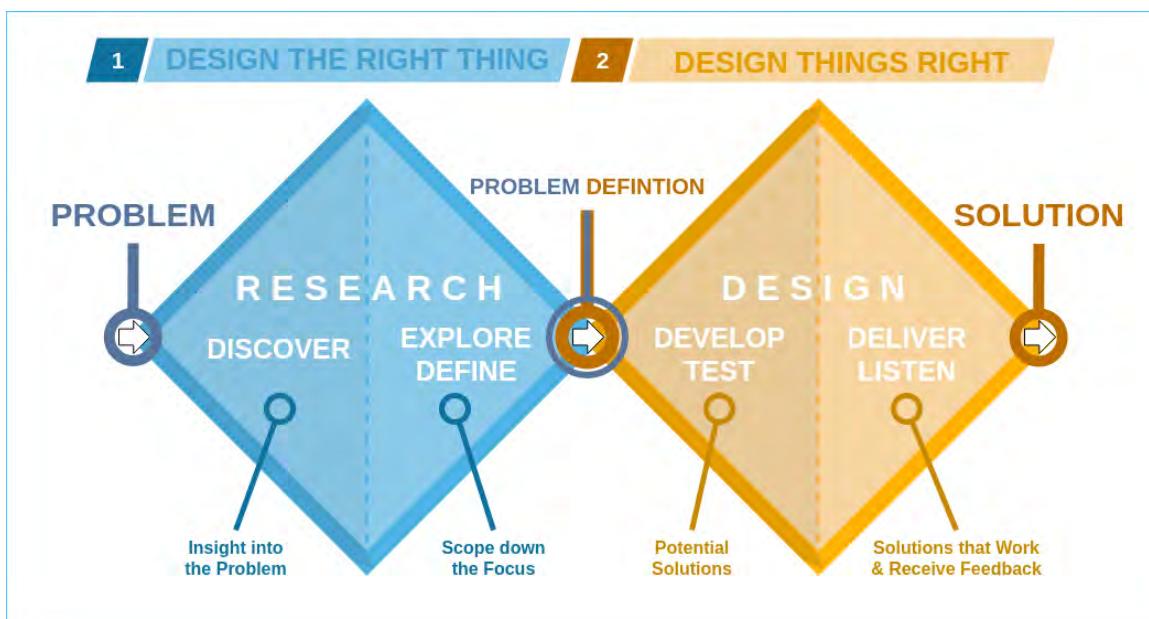


Figure 6.1⁸ “Double Diamond” design process model, popularized by the British Design Council, has two parts. The first need-finding phase is all about investigating *what to design* that will best make a positive impact in the problem space. Once the design goal is set at the *problem definition*, the second phase is all about prototyping towards an effective design.

⁸ Double Diamond Process Diagram by Digi-ark – Own work, CC0, <https://commons.wikimedia.org/w/index.php?curid=94113884>

Our problem statement for Part II of this thesis is: **design a better interactive version control for doing data science exploratory programming.**

On the onset, it's not clear what makes a *good* interactive version control for doing data science programming. *How will we know if we've achieved a good design?* To address the uncertainty of this design space, we actively prototype, build, and test interactions with data scientists. Since we are designing for a data-intensive and very specific technical context (**history in data science exploratory programming**), we found early on that there's a limit to how richly a person can *imagine* themselves in that situation. Thus our design process includes a mix of paper-prototypes (where the onus is on the participant's imagination) and more costly interactive functional prototypes where we actively place participants in a simulated data science context to observe what happens.

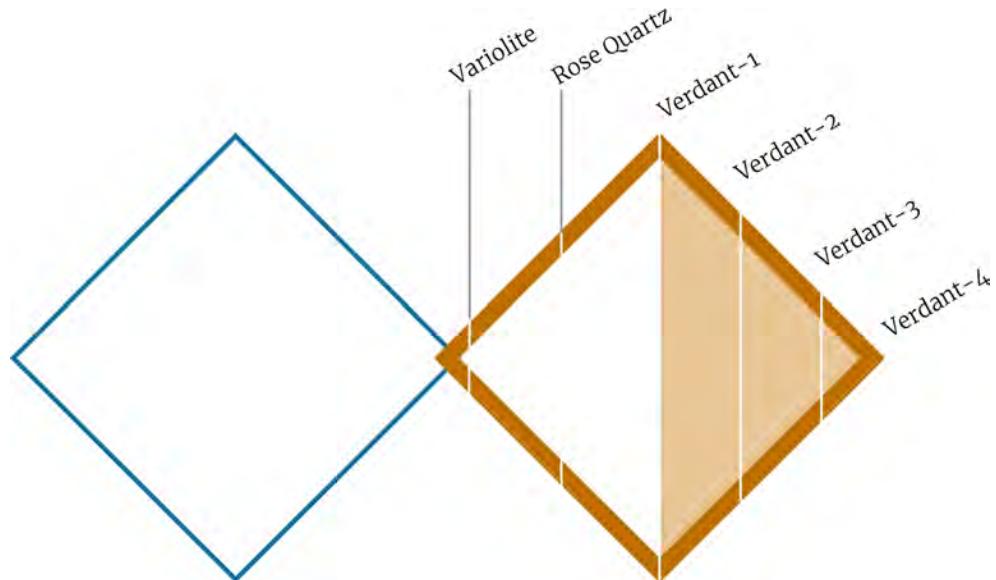


Figure 6.2 Progression of Systems

SYSTEM OVERVIEW

The “diamond” in this design process framing is characterized by an initial broad exploration, followed by a narrowing as the design team converges on which ideas best address the problem. Duly, early in the design process we prototyped **Variolite** (2016–2017) and **Rose Quartz** (2017) systems, which imagine very different workflows for how a person might interact with their experiment history. We then combine ideas from each to create **Verdant**, which we then iterate on in a product development process.

Shown in the table below, each prototype conceptualizes our design goal in different ways:

Scope Where is the user working?	Version Target What are we versioning?
 Variolite Chapter 7	1 Python script file in Atom editor Code snippets

 Rose Quartz Chapter 8	2+ script files in Atom editor, an instrumented bash terminal, and a Rose Quartz desktop app to display history	Code snippets Data files Parameters Outputs Notes
 Verdant Chapter 10–12	Jupyter notebook (<i>running in Jupyter Lab platform</i>)	Code cells Outputs Markdown cells

Initially in Variolite we aimed to formalize what we had observed of “informal versioning” where data scientists experiment with ideas by generating alternative versions of specific code snippets (Exploratory Programming Study, Chapter 3). Thus Variolite is built to help users manage code snippet versions *in-situ* within a Python script. Variolite’s conceptualization of history is extremely specific to the mechanics of *how* data scientists version during exploration.

In a usability study of Variolite (Chapter 7), we found that although Variolite’s interactions were well-received by users, the system’s design was insufficient to capture the full picture of what experimenting with code and data meant to users. What a data scientist considers as an “experiment” is really more about cause-and-effect, i.e., *when I make this change to code, data, or input parameters, how does that change the resulting output?* Our research goals shifted from supporting a specific kind of version interaction (informal versioning), to more broadly seeing if we can serve the underlying need of users to capture the key semantic information about their experimentation.

In Rose Quartz (Chapter 8) we took inspiration from Guo and Seltzer’s Burrito system [Guo & Seltzer 2012] and tried to collect experiment history from *the user’s workflow across their entire computer*. Our primary goal in Rose Quartz was to collect the full picture of code experimentation cause-and-effect, whether it be a change to data, input parameters, or code snippets, and whatever output might result. Moreover, we sought to find ways to visually *summarize* all of this experiment log data in a way that could boil down the key semantic details of experiments to a user. This idea, in concept, is similar to summarizing models as Model Cards [Mitchel et al. 2019] or summarizing data as Datasheets [Gebru et al. 2018] (see Chapter 5) in that Rose Quartz attempts to form loose log streams into easily digestible snapshots.

When that idea proved too lofty and unwieldy (Rose Quartz was subsequently abandoned), we narrowed to designing for a computational notebook context for our next prototype, Verdant [Kery & Myers 2018].

Although we needed to scale back our ambitions from Rose Quartz, a computational notebook scope allowed us in Verdant (all 4 versions discussed in subsequent chapters 10, 11, 12) to draw relationships among code changes, notes, output, and runtime information to form a reasonably solid view of a data scientist’s experimentation.

Although a computational notebook contains roughly the same content as a normal code script, a notebook breaks down that content into smaller cell chunks and makes the whole document interactive. Similarly, we were inspired to create an *interactive* version of familiar history

dynamics from Git, adapted to work at the level of cells and output rather than files and folders. The design framing of a kind of “miniature interactive Git” provided us with a rough expectation of how Verdant history should operate on the back end. By leveraging those familiar version control system design patterns, we were able to focus on designing interactive visualizations more specific to helping data scientists understand their experimentation.

Verdant went through many iterations (Verdant-1, Verdant-2, Verdant-3, and Verdant-4) over a period of 3 years. During this time, the first two iterations were research prototypes allowing us to explore a variety of different research ideas for how to support data scientists with history data. As we built and tested these prototypes and our designs matured, the later Verdant-3 and Verdant-4 refined these ideas into a more robust and reliable functional prototype for the real world.

CHAPTER CONCLUSIONS

Ultimately Verdant, and all our tool work in this thesis, are designs imagining a particular future: “What would happen if data scientists *had* excellent history support for experimentation? What would it look like? What would it enable them to do?” Our designs are hypotheses around what an excellent history support *might* look like and how it might function.

Crucially, since we are designing user interactions on top of complex data flows (history data), these data elements make for a more complex design process. Whereas a classic simple user interface design process espouses low-fidelity sketches and mockups to get user interactions right *before* functional prototype, in this setting we must design *the history data* and history backend approach in tandem to UI. Real history data is complex and messy. As covered in the following chapters, we use a combination of paper prototyping, mockups, and functional prototypes to design with active history data flows. During this iteration, the nature of the history data we collect and how it is collected, stored, and presented, all evolve as engineering efforts needed to match our evolving UIs. Due to the high cost of implementing designs into functional prototypes, each of our systems covered are engineered *just to the point* that they can be tested with real data practitioners to verify that our designs are going in the right direction. Often, at the point of these studies, designs that *had* tested well as mockups hit usability tangles in real usage, leading us to need to dramatically reconfigure design details for our next iteration.

Finally, to test our hypotheses, we ultimately invest in a substantial product development period with Verdant, beyond the demands of a research prototype, so that Verdant could withstand the stress of real exploratory programming work over a sustained amount of time. In [Chapters 10](#) to [Chapter 12](#) we describe in-depth the progression of Verdant. Finally in Section III of this thesis we conduct an exploratory study of how data scientists work in experimentation with Verdant, and how well Verdant achieves our goals for supporting easy-to-use experiment history.



Chapter 7. Interactive Snippet Versioning

Featuring Variolite

Research done in collaboration with Brad A. Myers and Amber Horvath⁹

INTRODUCTION

This chapter we present our first system design, Variolite, which attempts to capture the same value people find “informal versioning” of code snippets.

DESIGN GOAL: CAN WE MAKE INFORMAL VERSIONING OF CODE SNIPPETS MORE ROBUST?

How do people ideate through code? We found (see Exploratory Programming Study, Chapter 3) that individuals working with data often rely on *informal versioning* to keep track of alternative ideas they try. Informal versioning behaviors include copying code, keeping unused code, and commenting out code to repurpose older analysis code while attempting to keep those older analyses intact. Unlike conventional version control, these informal practices allow for *fast* versioning of any size code snippet, and quick comparisons by interchanging which versions are run. A simple use case is shown below:

```
model = RandomForestClassifier()  
#model = LogisticRegression()  
model.train(x_train)  
model.fit(x_test)  
score(model)
```

Above, the code author uses comment syntax `#` to toggle between two different model types. In terms of “good” programming style notions from software development, this would broadly be considered a sloppy way to test out model types. In better practice, data scientists can write a function that *parametrizes* the model elements the author wants to experiment with:

```
def do(model):  
    model.train(x_train)  
    model.fit(x_test)  
    score(model)
```

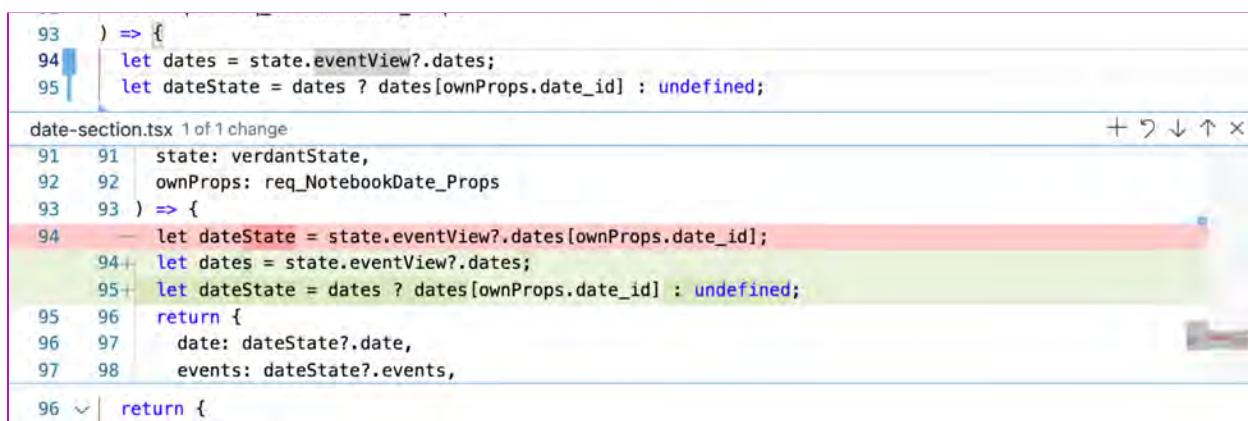
⁹ This chapter is based in part on the conference paper: Mary Beth Kery, Amber Horvath, and Brad A. Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In CHI, vol. 10, pp. 1265–1276. 2017.

However, we observe that authors often don't use this level of formality during exploratory work, where *quickly* navigating a broad decision space of possibilities takes precedence over neatness (Exploratory Programming Study, Chapter 3). The comment `#` toggle and other, admittedly slapdash, means of informal versioning are often used in exploration for speed. Instead of asking authors to *never* write sloppy code (which we argue is unrealistic), can we intervene in place of where authors would use a comment toggle, to provide a robust experiment pipeline for free? This is our design goal for Variolite.

PROTOTYPING PROCESS

Prototyping was conducted at the HCII at Carnegie Mellon University with the help of Amber Horvath and Brad Myers who contributed to the design.

We decided to focus on designing in-editor version support, since commenting and keeping code are both in-editor informal versioning methods. For inspiration we reference a rich ecosystem of in-editor version interactions for Git. For instance in Figure 7.1, the popular interactive development environment (IDE) Visual Studio (VS) Code has a variety of very nicely designed ways a programmer can interact with their Git versions in the editor. Although these well-established interactions already exist for Git, versioning for exploration presents a different enough use case that we cannot adapt those directly. Instead of dealing in the editor with *many changes between just two versions* (the current and prior), in experimentation practitioners deal with relatively *few changes among many versions*. In experimentation a user may want to see 12 different versions of parameters for their machine learning model—something which may constitute only a few very small text edits yet many versions they want to browse at once. In Git, a programmer typically only works with one version at a time (versus one other prior version) but typically makes larger edits across multiple files to constitute a single commit.



The screenshot shows a code editor in VS Code displaying a file named 'date-section.tsx'. The code is a function with several lines of logic. A specific line of code, 'let dateState = dates ? dates[ownProps.date_id] : undefined;', is highlighted with a red background, indicating it is a change from the previous commit. The code editor interface includes line numbers on the left, a status bar at the bottom, and a toolbar with various icons at the top right.

```

93 ) => {
94   let dates = state.eventView?.dates;
95   let dateState = dates ? dates[ownProps.date_id] : undefined;
96 
97   return {
98     date: dateState?.date,
99     events: dateState?.events,
100    ...
101  }
102}

```

Figure 7.1 An interaction in VS Code uses a box display to highlight changes around a specific code snippet since the last commit. This display inspired our variant box metaphor for managing many alternatives of a code snippet, rather than the most recent diff change.

To explore the design space, we first sketched a number of possibilities for interacting with versioning for experimenting within a code editor. We showed these to a convenience sample of 6 data scientists. Between each informal 15–30 minute session we iterated on the drawings based on the open-ended feedback we received. Earlier sketches focused on comment `#` toggle itself. For instance, in Figure 7.2 below, we proposed a code annotation approach, where users could label commented code with an experiment name like “RandomForest with x” and our system would automatically log usage of that code as that particular experiment.

```

function float Orange(data)
{
    results = Algorithm3(data, paramT);
    System.out.print(results.size());
    ProcessRes(paramA, results);

    /*
        results = Algorithm1(data, paramB);
        ProcessRes(paramE, results);
    */

    /* RandomForest with x
        results = removeOutliers();
        results = Algorithm3(data, paramB);
        ProcessRes(paramA, results);
    */

    return results;
}

// RandomForest with x import data_testA

```

Figure 7.2 An early concept sketch focused on the idea of informal versioning via commenting and uncommenting chunks of code. We used a “layer” interaction metaphor: like the layers of an image can be toggled visible/invisible in a drawing program like Photoshop¹⁰, a user could use the layer pane to toggle which versions of their code were active as commented/uncommented. This idea was discarded because it scales poorly when there are more than a few versions involved. It also still leaves the clutter of commented-out code in the user’s source file, which is arguably a major flaw of informal versioning via commenting in the first place.

However, in practice we found that, since the comment `#` toggle is considered a sloppy form of hack-y versioning, some data scientists were not a fan of exacerbating its usage. Based on feedback and the issue of scaling up to 10 or so versions per chunk of code, we narrowed towards designs that more closely resemble the in-line version box view in VS Code in Figure 7.1(1).

¹⁰ Adobe Photoshop <https://www.adobe.com/products/photoshop.html>

From there we iteratively designed and implemented our adaptation of a version box in a prototype tool called Variolite¹¹.

Variolite is implemented in CoffeeScript (a Python-esque dialect of Javascript), using the Atom editor's package framework¹². Atom is an open-source code editor developed by GitHub, first released in 2015. It has over 1 million active users, and is close in style to other more mature editors such as Sublime Text, which are popular among Python programmers. Like most version control tools, Variolite is language agnostic and can be used with any programming language or plain text. Here we show examples in Python, as it is a popular language for data science tasks.



Figure 7.3 Variolite's “variant box” interaction. In (1) the user highlights a region of code with their cursor that they are interested in experimenting with. In a right-click menu option they select “wrap in variant”, and in (2) a variant box appears above their selected code. The user now has the option of adding variants to this code, which they do by clicking on the variant box header and selecting “new branch” (3). In (4) the user has created a new variant for their code. They can switch between their variants by clicking the variant’s title.

¹¹ Stands for: Variations Augment Real Iterative Outcomes Letting Information Transcend Exploration. Variolite is a kind of volcanic rock.

¹² Atom editor <https://atom.io/>

SYSTEM: VARIOLITE

Our core interaction in Variolite is intended to be (nearly) as lightweight as commenting code, while providing much more power and robustness. Users of Variolite draw “variant boxes” around regions of code (analogous to a block comment), where the code within the box can then be locally versioned or branched (Figure 7.3). As with commenting, where a user can use the comment symbol as a switch to control execution, users can control which version is run by a simple switch of the active tab on the variant box. This is similar to the tab-based versioning shown in Juxtapose [Hartmann et al., 2008]. However, whereas Juxtapose shows alternatives of a file in the code editor, we extend this interaction to encompass any amount of code. A user can draw a variant box to create alternatives of a file, a group of functions, or a single line of code. Multiple variant boxes can exist in a file and they can be nested (Figure 7.4 b,c). Variant boxes can be created or dissolved back into flat code as needed. Variant boxes are always present and visible in the document structure. In order to not impede code readability, when a variant box is selected the full box appears more sharply with a menu of buttons (7.4 c) and when not selected it fades to faint labels (7.4 b).

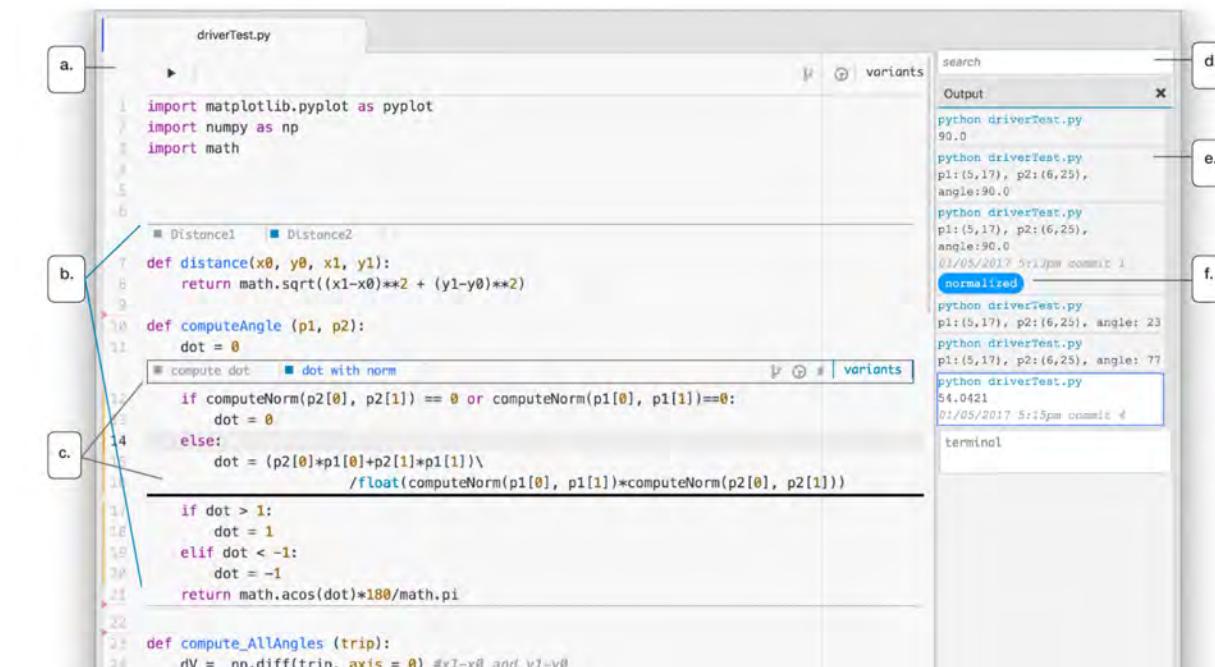


Figure 7.4 Variolite with labels for different features. (a) the top level variant box that wraps the entire file and also acts as the tool menu. (b and c) two different variant boxes, one nested within the other. (d) a search bar for finding outputs and versions. (e) the output pane and (f) where the user has given an output and its commit a custom tag.

Like informal versioning, our intention for Variolite is to provide a simple structure that is sufficiently flexible so programmers can leverage versioning in whatever way they need during their exploratory process. Similar to a more formal experiment pipeline, a user can put variant boxes around features, options or other parameters included in a model. Now, by simply interchanging which versions of each of those alternatives are run, they can explore different

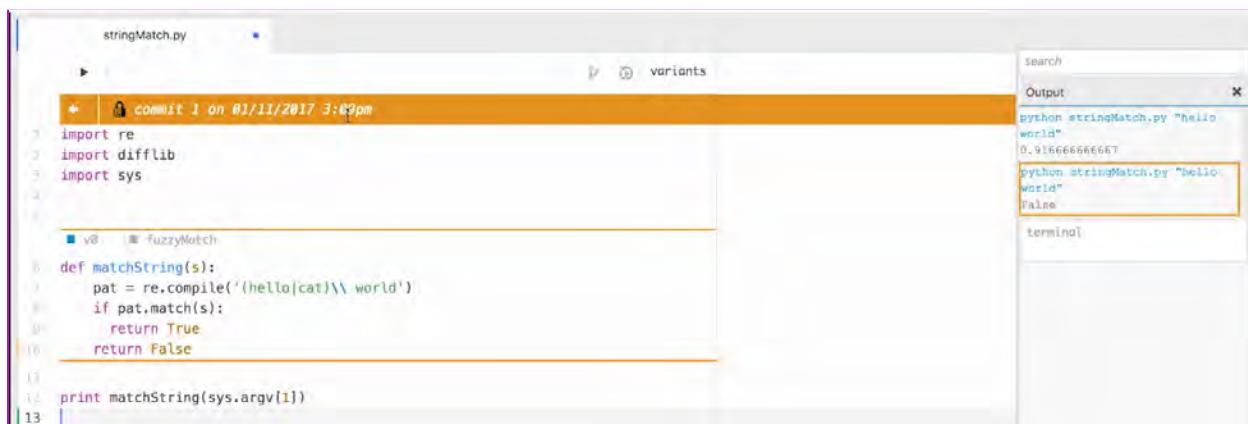
combinations of the different things they have tried. Instead of a purely linear iteration, they can re-try features they used in the past with new versions of the algorithms' parameters.

Logging Experiments for Reproducibility

In order to provide a full experiment logging pipeline for the user, we add UI elements that allow users to run code directly from the Atom editor (Fig 7.4 a & e) such that Variolite can collect inputs and outputs.

Each time the code is run, Variolite creates a snapshot of the code file and automatically records the parameters used, variants used, and all inputs/outputs from the run. This provenance data is saved in JSON format separately from the code. Variolite visualizes experiment results in the output pane (Figure 7.4 e). To help users find interesting results among many runs, a user can add custom tags to versions that are promising (Figure 7.4 f). Finally, to help recover results from longer sessions, Verdant includes a search bar to search for results by keyword or tag (Figure 7.4 d).

Double-clicking a given output in the output pane (Fig 7.4 e) causes Variolite to set the entire code back to the past version of the file and the past version of each variant box that produced that result. If a new variant box was created later in time after that output, it will not appear in this view. While viewing an earlier commit, the user cannot edit the code, but they can re-run it, copy it, or create a new branch from that point in time. Variolite keeps past commits as immutable to preserve the output history. If a user creates a branch from a past commit, however, they can continue editing from that point.



The screenshot shows the Variolite interface. On the left is a code editor window titled "stringMatch.py" containing Python code. The code defines a function `matchString` that uses regular expressions to check if a string contains either "hello" or "cat" followed by "world". It also prints the result of the match. The code editor has a yellow border around the entire window, indicating it is a past view. On the right is an "Output" pane showing two command-line entries. The first entry is "python stringMatch.py "Hello world"" with output "0.916666666667". The second entry is "python stringMatch.py "Hello world"" with output "False". The second entry is highlighted with an orange border, and its output is also outlined in orange. A "search" bar is visible at the top of the output pane.

```

stringMatch.py

commit 1 on 01/11/2017 3:09pm
1 import re
2 import difflib
3 import sys
4
5 v8 = FuzzyMatch
6
7 def matchString(s):
8     pat = re.compile('^(hello|cat)\\ world')
9     if pat.match(s):
10         return True
11     return False
12
13 print matchString(sys.argv[1])

```

Figure 7.5 Variolite showing an earlier version of the entire code file. When a user double-clicks a given output (outlined in orange) Variolite shows a read-only view of the code that generated that output. The entire editor is framed in orange to indicate that this is a past view of the file.

STUDY: TESTING THE USABILITY OF VARIANT BOXES

Our goal at this stage of the design was to create a reasonable alternative for informal versioning interactions, such that in tool form, issues around informal versioning become more tractable to design interventions. Thus, we conducted a usability study. To test if our core interaction for Variolite is sound, we only tested the variant box interactions, and did not have

participants engage with interacting with past output, search, or other features. Some specific visualizations, described below, were added following the study based on participant feedback.

We recruited 10 participants, a mixture of undergraduate and graduate students (7 male, 3 female). Participants had on average 5 years of programming experience and 1.5 years of experience with data analysis. The pilot study was conducted in our lab, using a designated MacBook computer. After signing a consent form, each participant was given a brief tutorial on Variolite, showing how to wrap code in a variant box and create a new version. Next, participants were given an Excel file dataset and a set of “exploratory questions” to answer about the data using the tool and a Python script. We gave participants fixed questions, instead of allowing them to freely explore the data, because this allowed us to focus their work on questions that built off of previous questions and required some versioning. After the coding task, each participant filled out an online questionnaire to give feedback on the tool, and was compensated \$20 for their time.

9 of 10 participants were able to successfully wrap code in variant boxes, create new versions, and switch between versions during the coding task. The one participant who struggled with the tool became confused when instead of manually selecting all the code in a function, she only selected the function name before using the command “wrap in variant”. She expected the tool to then wrap the entire function in a variant, but instead it only wrapped that single line. By adding language-specific static analysis checks to Variolite, a future iteration of the tool could include scoping rules such that if the user wraps the line def foo(): in Python, this would appropriately wrap the whole function.

As recommendations for new features, the participants requested better ways to distinguish different versions. Several mentioned the ability to name their versions was a very useful feature, but more automatic techniques were requested. This motivated some of the features we added to Variolite next, including ways to navigate and search the branches and commits, and cues such as tags. One participant was concerned with becoming overwhelmed with too many versions of a part of the code, which motivated the branch view (Figure 7.6).

Overall, 9/10 participants reported on the questionnaire that they liked the tool and found it easy to use. All 10 wrote that they would consider using it in real life, and one participant even emailed us after the study asking when the tool would be released to use.

DESIGN ITERATION

Although participants reacted favorably to the variant box interaction and idea of Variolite, scale was an issue that worried participants (and us too as designers). The tab layout at the header of a variant box is not manageable to switch among more than 3 to 4 different branches because of the limited space. A list of 10 to 20 branches in a variant box may look overwhelming.

We began prototyping more advanced visualizations to help users navigate through possible versions and variants in a variant box. Shown below, by clicking on the revision tree icon, the user can access a larger revision tree showing all the branches of that box (Figure 7.6), such that they can control which 3 or 4 branches are actively showing. See Chapter 5 for a detailed

overview of the revisions tree visualization, which we borrow from classic software version control systems like Git.

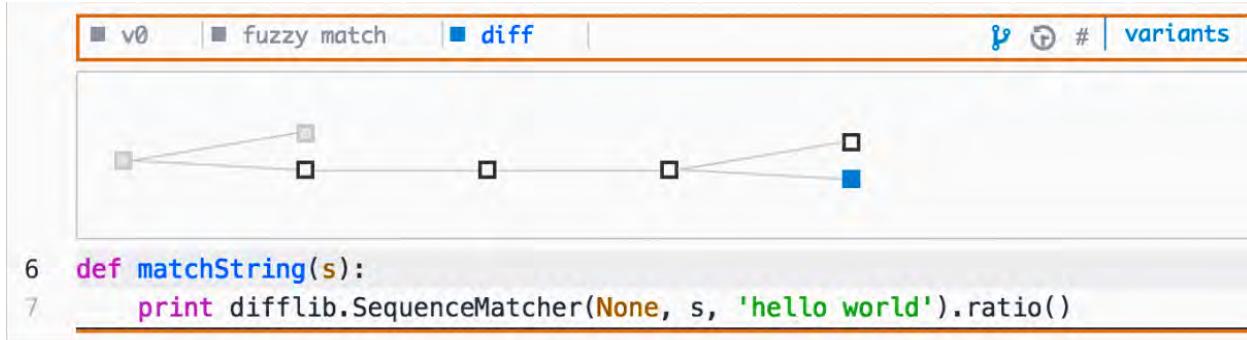


Figure 7.6 Navigating branches

If a user wishes to backtrack a single variant box to an earlier state, they can also navigate its commit data by clicking the clock icon (Figure 7.7). This activates a timeline slider, where (similar to video editing software) a user can scroll the slider back to view the code at different commits. The user can use this form of time travel both at the file level and with individual variant boxes. The same backtracking can be achieved by clicking on individual commit nodes in the revision tree view in Figure 7.6.

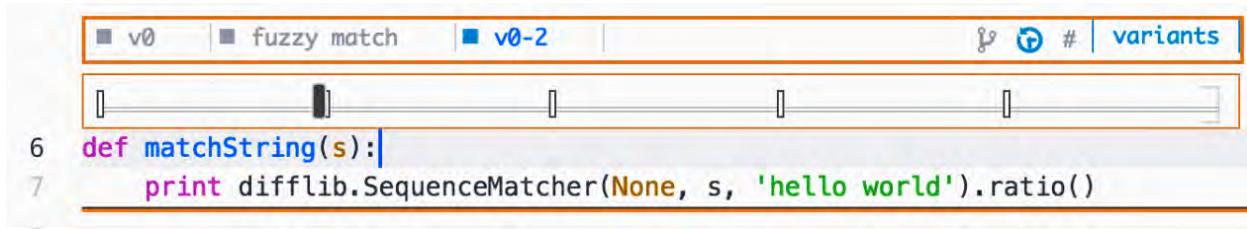


Figure 7.7 Navigating to a past commit. The orange color of the variant box indicates that the code is showing a past state.

Although in Variolite, users can name different versions, our study of data scientists suggests that this is not always done, and even if so, they might use similar names which can be hard to distinguish. Furthermore, Variolite cannot force users to pick names that are logical or easy to distinguish. Thus, to address the problem of distinguishing versions, we use metadata to provide users with a variety of different context clues. This decision is informed by prior work by Ragavan et al. [Ragavan et al. 2016] who used an information foraging theory approach to study how programmers distinguish between similar versions. Distinguishing versions can be a difficult task, and participants in that study leveraged a variety of code cues such as file name, output, data, and code features.

Some metadata is similar to what is provided by a VCS. For example, a user can name each branch. Each commit and each branch shows a date when it was last edited. Each variant box can show a revision graph, so the user can see the order and relationships among branches. Variolite also gives additional cues:

- The ability to tag any branch or commit with a custom tag, for example “Paper version”, “Nice graph!”, or something task-specific like “Crows distance” (Figure 7.4 f).

- The ability to search in past outputs, branches, and commits and not just in the current file (Figure 7.4 d).

Ultimately, however, we began to worry that these new interactions, while mitigating the complexity of interacting with variant boxes, were a “bandage” over a more serious underlying problem. Local versioning techniques, like the venerable comment `# toggle, don't scale well` when used in abundance littered all over a code file for long periods of time. This problem framing of local versioning as a kind of *configurable* experiment pipeline within a code file is close to Software Product Lines used in industry to mark multiple configurations of code within a code file. These are known for their complexity [Clements & Northrop 2001] and having files with variability is a known usability challenge.

LIMITATIONS OF HISTORY MODEL

During design we recognized that it is insufficient to log a code experiment as just the snippets the user changes in a variant box. These code snippets (e.g., `if computerNorm(p2[0]...`) are typically not complete programs to run in isolation and depend on other parts of the code file. To be able to log an experiment such that the user can later reproduce it, we need to represent the entire code file. Here we heavily borrow from conventions in Git, and record a full *commit* of the code file each time the code is run. The chief difference we introduce in Variolite (and continue to use in the rest of this thesis) is the idea of a *hierarchical* file history. That is: we record a whole code file in a series of commits as v1, v2, v3 ... but that code file contains specific parts of interest, here the variant boxes, which *themselves* have history. In Variolite, each variant box has its own revision tree of commits and branches, like a typical VCS. However, rather than a single revision tree existing for the entire code file, Variolite models the file as one revision tree that points to child revision trees for each variant box in the code. Variolite keeps a revision history for the file, and then links this revision tree to child trees of any variant boxes that are created in the code.

As demonstrated in Fig 7.8, having different entities, each of which has their own history, leads to a complex history datastore. This datastore is based on the history structure shown in Variolite in Figure 7.4. The file `driverTest.py` contains `variant box (b)`, while `variant box (b)` contains `variant box (c)`. However, `variant box (c)` only occurs in the `Distance1` branch/tab of `variant box (b)` and wasn’t actually added by the code author until commit 1 of the `Distance1` branch/tab of `variant box (b)`, which happens at the same time as commit 3 of `driverTest.py`. Meanwhile the timestamp, outputs, and other metadata recorded for an experiment by Variolite maps onto specific combinations of commits/branches of all the constituent program parts. Suffice to say, even with only a few commits logged, this history datastore is complex and unwieldy.

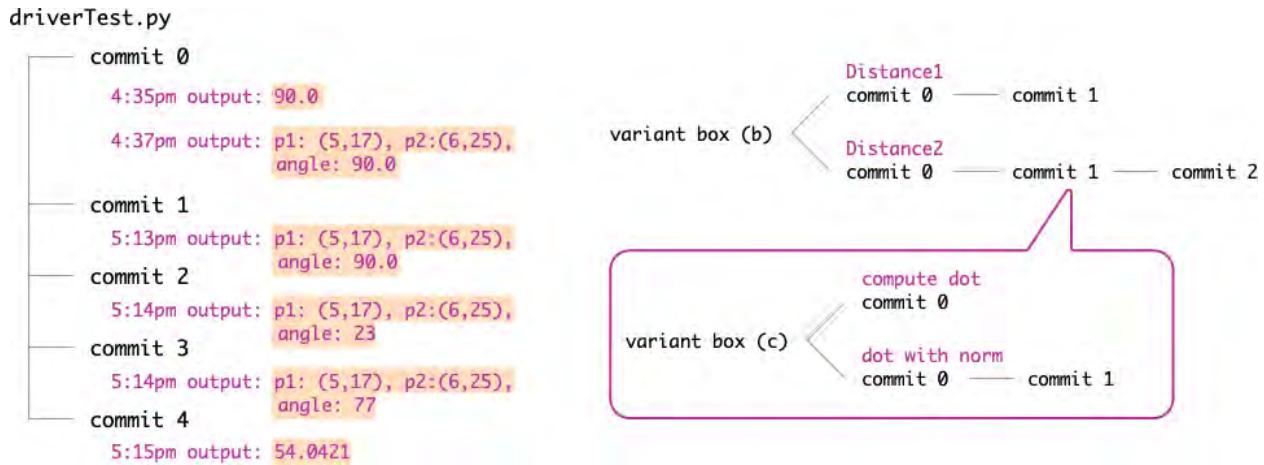


Figure 7.8 A diagram showing how variant box histories relate to each other and the history of the entire code file `driverTest.py`.

The model for history relationships in Variolite is problematic and can quickly devolve into a combinatorial nightmare. However, we nonetheless believe that finding effective ways to model detailed relational histories between specific code snippets, input, data, and output is worth continued research, which we carry forward in this dissertation.

CHAPTER CONCLUSIONS

In this chapter we designed Variolite from paper prototypes up to a functional prototype, tested it's usability, and then iterated further on the system design up to a point at which we decided that the core interaction model Variolite uses is unsustainable. Although this design cycle ended with the need to pivot to new history ideas, the work we did in Variolite was still ultimately extremely helpful in pushing forward our understanding of what history tooling should look like for experimentation. Here I try to summarize these design findings:

- + **Support specific fine-grained versioning:** Allow creation and access of multiple alternative versions of individual snippets within a code editor. Although we posed this need for support as a hypothesis back in Chapter 3, highly positive feedback on Variolite from both users and the broader research community gave us validation that practitioners want this functionality.
- **Keeping versions in UI tabs does not scale well:** While a tab interaction appeared compelling and easy-to-understand during prototyping, we found during usability testing that users felt overwhelmed as more versions and more tabs were added. Due to limited screen space, we could only display 3-5 named tabs, depending on how long a name each version had. While we proposed work-arounds like the revision tree visualization shown in Figure 7.6, we no longer feel that tabs are the best interaction metaphor for realistic version situations.
- + **Support continually shifting focus:** In prototyping Variolite we had this idea that participants would have specific regions of their code that they would want to vary. This assumption came from our Exploratory Programming Study in Chapter 3, where we observed practitioners informally version specific important regions of their code.

However, a limitation of that study is that we saw practitioners' artifacts from just one point in time. In the Variolite usability study, by watching people work over time, we learned that these regions of interest a user wants to version *continually shift over time* as the user tests out different hypotheses and develops different ideas. Also importantly, the regions that a user wants to version are not necessarily continuous regions of code, but could be related to snippets scattered around the file. Variant boxes in Variolite were designed to provide history to a specific fixed code region, and we learned that a more flexible history model is needed.

- + **Version all or nothing:** The issue of practitioners continually shifting focus in their file means that they want history of unpredictable regions of code for which we may or may not have collected historical data in a variant box. Additionally, as discussed in our limitations section, versioning just *parts* of a code file leads to an ugly combinatorial problem. Versioning just parts of a code file thus gives us an overly complex history model that still cannot meet participants' constantly shifting history needs. From this we learned that **the safest approach to avoid these issues is to just version everything**. If we version *all* of a user's code file for each experiment, we can later retrieve the history of *any* specific portions of the file the user wishes, while simplifying the structure of the history we store.



Chapter 8: Capturing the Full Picture of an Experiment

Featuring Rose Quartz

Research done in collaboration with Brad A. Myers, Marissa Radensky, Mahima Arya¹³

INTRODUCTION

In this chapter we switch focus to a different part of the design space of history tooling. Instead of looking at editor interactions to support history as someone experiments, we zoom out to prototype interactions to capture the full workflow of an experiment that may occur across any number of tools a practitioner uses for their work. The system of this chapter, Rose Quartz, ultimately reaches a dead-end, but still provides important insights into the design space.

DESIGN GOAL: CAN WE CAPTURE A FULL DATA EXPERIMENTATION PROCESS?

As we continued to iterate on Variolite, it became evident that while we *could* create a polished incarnation of Variolite that overcame the limitations of original variant boxes, we would still be capturing just a piece of experimentation rather than the whole workflow. For code experiments, data scientists work in code editors, but also data files, terminals where they execute scripts, personal note files, spreadsheets, and more. How do we capture the cause and effect of, for instance, a model experiment, if its inputs and outputs are beyond what we can record in a code editor? While Variolite's run button and imitation-terminal UI sufficed for a prototype, it was more a UI "patch" to allow us to experiment with designs including input/output data, rather than a realistic solution.

We aimed to push our research further. Rather than focusing on specific informal versioning mechanics like the comment `#` toggle, which are more symptoms of an underlying user need, we began to pivot into understanding "an experiment". We reframed our design goal from providing users with interactions "to achieve experiment versioning" to instead providing users with interactions "to understand and learn from their experiments", with an increased technical focus on automatic version collection and how to present versions to users in a usefully synthesized way.

Our next system design, Rose Quartz¹⁴, did not make it past the paper prototyping and early prototyping phase of implementation, but is worth mentioning here for lessons we learned from it.

¹³ Work in this chapter was never published.

¹⁴ Rose Quartz is named for a type of quartz and stands for: Recollection Of Serial Experiments: Quiet Utility Always Recording Tree-structure Zen.

DESIGN PROCESS & LIMITATIONS

Prototyping was conducted at the HCII at Carnegie Mellon University with the help of Marissa Radensky, Mahima Arya, and Brad Myers who all contributed to the design.

Rose Quartz is heavily inspired by Guo and Seltzer's Burrito system [Gou & Seltzer 2012]. The spirit of this work also aligns with formats for summarizing models such as Model Cards [Mitchel et al. 2019] or formats for summarizing data such as Datasheets [Gebru et al. 2018] although that related work had not yet been published at the time of our design process. The main idea of Rose Quartz is to take information about an experiment coming in from multiple sources, and summarize that workflow in one place as a “card”. For instance: a data scientist uses a terminal to run a python script, which outputs a plot in an external plot viewer. There are at least three sources involved in that workflow: 1) the command and parameters in the terminal, 2) the python script in a file, and 3) the plot in an external tool. Rose Quartz could stitch back together those bits of information into a coherent “experiment card”. A similar idea had been broached in the Burrito system [Gou & Seltzer 2012] in the Computational Context Viewer shown below in Figure 8.1.

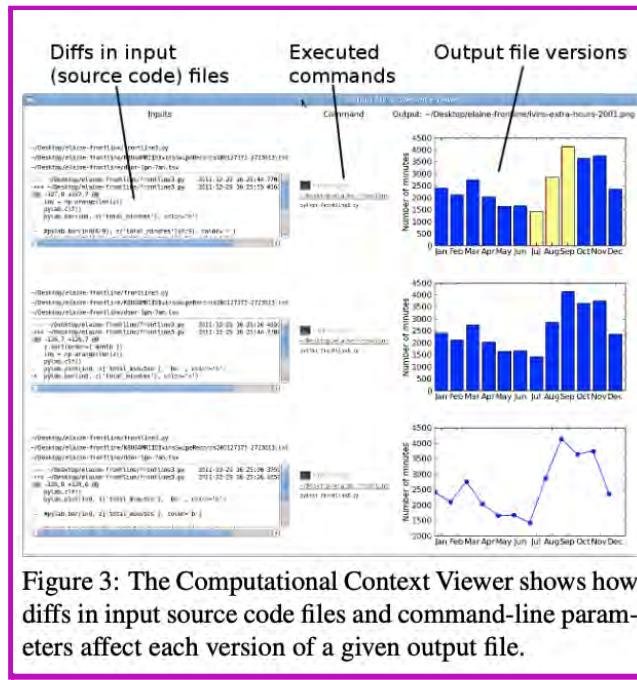


Figure 8.1 A view from the Burrito system that attempts to summarize code experimentation [Gou & Seltzer 2012]

We believe there is much more design work to be explored in this space, and we were particularly interested in how users might understand a series of experiments connecting together into a coherent narrative. Ways of automatically connecting workflows among the many disparate systems a data scientist uses is also of interest to the broader community [Kim et al. 2017]. However, these ideas we leave to future work, due to major data and technical barriers to the design process, discussed next.

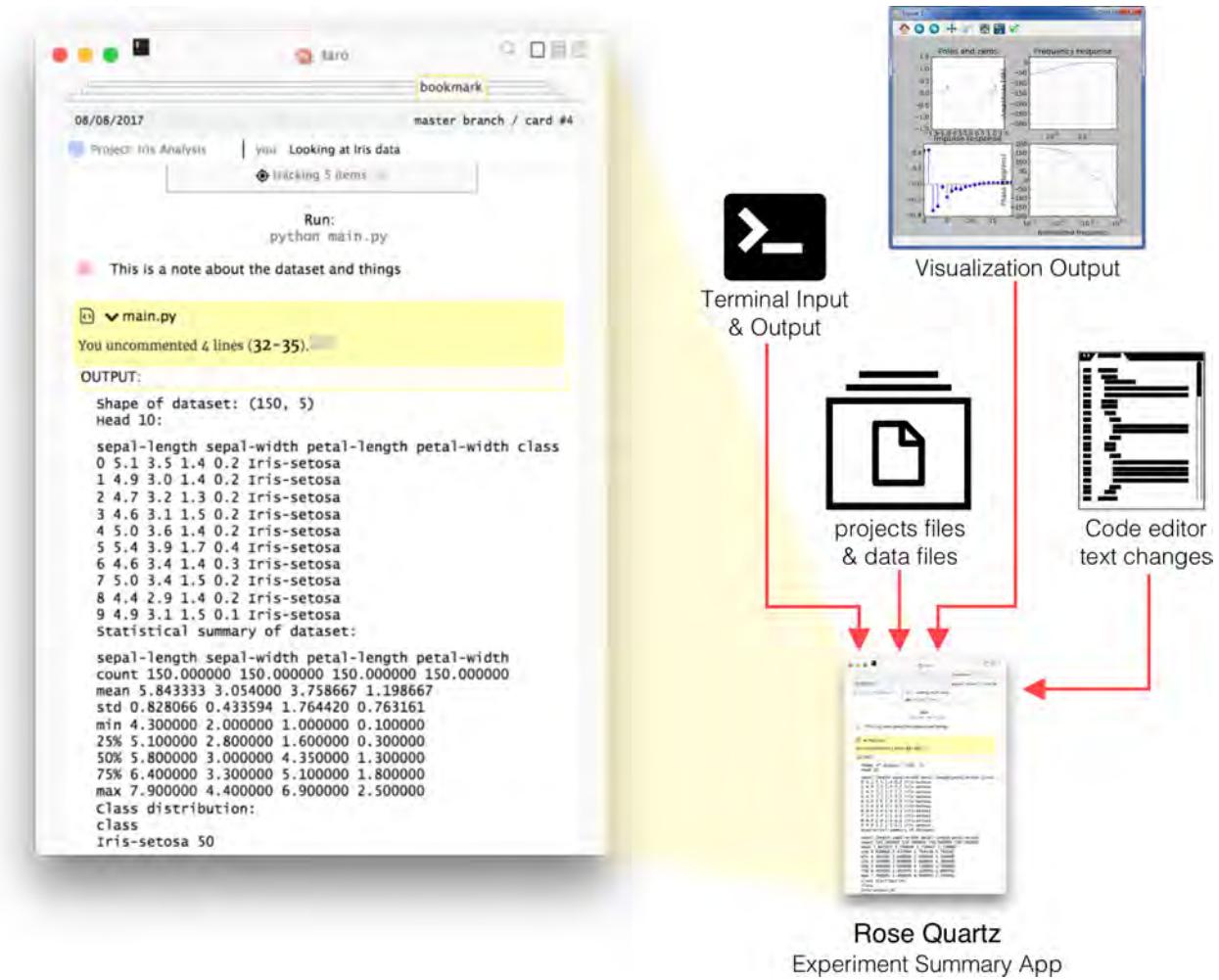


Figure 8.2 The main interface of Rose Quartz which, unlike Variolite, was meant to sit in a separate window from a practitioners’ code editor and passively and automatically collect versioning information from any code editors or tools the practitioner was using in their workflow (shown on the right). Rose Quartz uses a motif of *cards*, where each single experiment is shown as a card, and a series of experiments recorded over time is shown as a stack of cards. The card shown at left details the code file run, the specific change made within the code, the output of the experiment, and any user created notes on the experiment.

The technical requirements of Rose Quartz quickly became a barrier, since to automatically collect *all* changes and outputs relevant to an experiment would require instrumenting different kinds of tools in the user’s workspace to collect all relevant information. While not impossible, this had a high engineering time cost. A related kind of automatic experiment feed had been done in Gou et al.’s Burrito system [Gou & Seltzer 2012], which had required them to instrument the root of a Linux OS itself to gather the necessary experiment information from different tools. More crucially, we were also discovering a major barrier to iterating on our designs.

The card design of Rose Quartz (Figure 8.2) represented one possible way to display the summary of a single experimentation. However, to test with users whether or not our visualizations effectively summarized experiments in a way that was helpful to users, we needed real (or at least realistic) experiment data to show. Given our findings in our Exploratory Programming Study (Chapter 3) that practitioners do not (and cannot realistically) capture their experiment history data at the level of detail we sought to automatically collect for them, it was not surprising that we were unable to obtain detailed experiment history from either practitioners or from projects on Github. As a temporary work-around we generated some experiment history as a design team by doing a simple data science project ourselves and manually recording all experiment data after each and every run. However, this exercise only exacerbated our need for real data, as it illuminated some challenging properties of what realistic experiment version data looks like:

- **Anticipate a High Volume of Versions:** When you record how often a person runs their code during exploratory iteration, even a simple task can accumulate hundreds of runs per hour. Given this volume of small “experiments” it is crucial to design for scale.
- **Anticipate High Version Redundancy:** Given that experiments often constitute a small change, there may be many experiments that are all related or overlapping in what the experimental change was. Since versions may look alike, it is important to highlight differences for the user.
- **Highlight Relevant Versions:** It turns out not all runs of code are worthy of the title “experiment”, as the user may need to stop and debug code or write print statements that do not carry much experimental value. It is important to surface which experiments carry meaning to not overwhelm the user with noise. Unfortunately it can be very challenging in practice to automatically identify meaningful edits, since importance is task-dependent and subjective.

Without the data we needed to design with, and with too high an upfront engineering cost to obtain it for Rose Quartz, we decided to pivot towards new system directions that would lower data collection barriers and get us more quickly to designing with realistic data.

CHAPTER CONCLUSIONS

Later in this dissertation, with the implementation of Verdant iterations 1,2,3, and 4 we do achieve effective data collection for realistic data, which opens back up Rose Quartz’s design avenue for future work. Like the previous chapter 7 with Variolite, this chapter again concludes on the point of needing to pivot our design process to a new approach. Again, though, we argue that although this might be seen as a “failure” for the specific prototype, Rose Quartz’s design cycle is nonetheless successful in moving forward our design agenda in this dissertation by teaching us new important aspects of the design space. Some of these, what realistic version data looks like, are already highlighted above. I will add a few more design takeaways here:

- + **Help users see the relationship between code and non-code artifacts:** An “experiment” is often a combination of changes made to code, data, or input parameters, with results in forms of output, data, or notes. While we know this to be true from prior literature and our own studies (see Chapters 3 & 4), prototyping with Rose

Quartz allowed us to see the code and non-code artifacts together that form an experiment. It becomes apparent that simply *showing* all an experiment's code, parameters, notes, and output artifacts to the user is not always sufficient for a user to actually understand what the experiment was about. It is important to point out for the user cause and effect relationships by highlighting what about this experiment changed in the inputs and outputs.

- **Capturing “an experiment” from different tooling sources is extremely difficult, but choosing a single central artifact-rich tool environment is far easier:** Since data workers work in a variety of languages, tooling environments, and domain-specialized data analysis tools, instrumenting *every possible tool* that a data worker might incorporate into their analysis workflow is a enormous undertaking. On the flip side, listening to every tool raises serious privacy concerns and the major sensing issue: how do we know when to record experiment history? Can we tell if the user is actually experimenting versus doing other activities on their computer? These problems were apparent early in the design of Rose Quartz. Although we moved forward prototyping Rose Quartz with just a few representative tools (code editor, terminal, plot outputs), we were also investigating tools or platforms that already contained multiple parts of a workflow. A centralized “hub” type of tool allows us to scope what work our history system is listening to. This is what led us to study computational notebooks in Chapter 4. Computational notebooks provide a sandboxed workflow and are far more tractable for sensing, since we can scope data collection to just what the notebook API provides us in terms of the user’s code, output, and markdown. Generalizing this point, other tools that encapsulate a workflow, or serve as a launch-pad for tools used by a company or organization, could serve similarly as a centralized middleman for sensing what tooling a data worker involves in their experimentation.

For those interested in pursuing the design of tools like Rose Quartz, the privacy and data collection challenges remain the biggest barriers. In terms of design, our work in Verdant does provide rich fine-grained data about experimentation. Although Verdant collects data inside of a notebook, this data would still be an appropriate starting point for designers wanting to prototype with what real experiment history looks like.

Chapter 9. Collecting & Modeling History in Notebooks

INTRODUCTION

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 [Codoban et al., 2015]. Data science programming relies on working with a broader range of artifacts: the code itself, important details within the code (Exploratory Programming Study, Chapter 3), parameters or data used as input by the code [Patel 2010], visualizations, tables, and text output from the code, as well as notes the data scientist jots down during their experimentation [Brandt et al., 2008]. The conditions under which code was run and under which data was processed gives meaning to a version of code [Patel 2010]. 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.

With an aim to explore experiment history from a HCI perspective, we care less about the technical challenges of acquiring history data (see Chapter 8) and more about designing with that data. For this reason, computational notebooks, like Jupyter notebooks, are an ideal environment for us to prototype our ideas. Computational notebooks are interactive environments where data scientists often conduct exploratory programming (Notebook Usage Study, Chapter 4). Crucially, notebooks contain a variety of *artifacts* related to experimentation, including code snippets, output both textual and visual, markdown notes, and the runtime environment itself. For the purposes of modeling how a data scientist might better use experiment history, notebooks provide us simple access to much of what might constitute an experiment, as well as an already interactive setting to test out ideas for interactive history.

In this chapter, we detail the system specifics and trade-offs of collecting detailed experiment history data in a notebook. This history model is used in all following systems of the dissertation, Verdant-1 through Verdant-4. Key design questions we cover are:

- How do we best structure history for answering users' history questions?
- At what **granularity** should we collect history? Finer-grained history will allow our system to answer more detailed history questions, but at the cost of more storage.
- What is a **meaningful version** of a user's work? Can we use clustering, filtering, and heuristics to collect history in such a way to ensure versions have value to the user?
- How do we answer code questions? We discuss the tradeoffs between versioning at the level of **code cells** versus the **abstract syntax tree (AST) structure of the code**.

MAKING HISTORY CONTENT-ADDRESSABLE

To let data scientists ask cause-and-effect questions about their experimentation, we would like to make history *content-addressable*. We envision a user directly clicking on an artifact in their work (or another person's work) to ask about its history: *How did I reach this hyperparameter value? What feature sets did the author try? What did this plot look like before with a different scaling?*

To make history directly addressable by artifact, we want to decompose a notebook down into a set of discrete artifacts that users can look up from our history database at runtime. Alternatively, given a set of ordinary file versions, one could design a history system to infer the history of specific artifacts (similar to Git blame) at runtime. We decided not to pursue this inference approach because the computational cost at runtime might hamper interaction speeds. Instead, we aim to design a reasonably efficient history database that stores versions such that history can be quickly retrieved relative to any artifact in the notebook.

Next we need to decide: what is an artifact anyway?

We would like an artifact to be a *meaningful semantic unit of work* within the user's document. To one extreme we could version at the character level (e.g., decomposing a word into letters 'w', 'o', 'r', 'd'). However, there doesn't seem to be any obvious use cases why a user would ever want to look up the history of a letter. The finer granularity at which we segment the notebook, the more history-having entities we need to keep track of in our database. Thankfully, notebooks offer cells as a clear unit of meaning.

A computational notebook is a modular document structure composed of an ordered list of cells. A typical notebook is shown below, that starts with a Markdown note (often used as the title and a note about the purpose of the notebook), followed by some code cells and output:

Markdown cell 1	Code cell 1	Code cell 2	Output	Code cell 3	Output
-----------------	-------------	-------------	--------	-------------	--------

We define a notebook as **N** where cells are named by their type: **M** for markdown, **R** for raw¹⁵, **C** for code. We name output by the cell that created them, e.g. **C3.01** refers to the 1st¹⁶ output of the 3rd code cell. At any point in time, the notebook holds a list of cells:

$$N = [M1, C1, C2, C2.01, C3, C3.01]$$

For the purposes of experimentation, we aim to allow users to look through history specific to important pieces of the notebook document such as *the code that sets up model parameters, that*

¹⁵ Note that raw cells (**R**) are very rarely used or seen in real life usage, but Jupyter Notebooks includes them as one of 3 core cell types. A raw cell is raw plaintext, that cannot be run or compiled into anything else. Every single time a raw cell has appeared in our own user studies, we observed it was created by the user by accident as a result of a mistyped keyboard shortcut.

¹⁶ Although we largely ignore this complication in the scope of our research, the output of a cell is actually a list of outputs **01,02,... on**. In our prototypes we treat the full list of outputs as a single output **01**, however there are cases in which a user might want to treat outputs individually. This is most apparent in long-running model processes for something like an image model which will progressively append one image iteration after another to the output over the duration of several minutes or longer.

one plot, or the metrics for the model. To do this we want quick lookup of all versions of specific artifacts. Just like we discussed hierarchical and relational history in Chapter 7, here we start with a definition of artifact-based history, where a notebook's history is comprised of cells and outputs, each of which carry their own history:

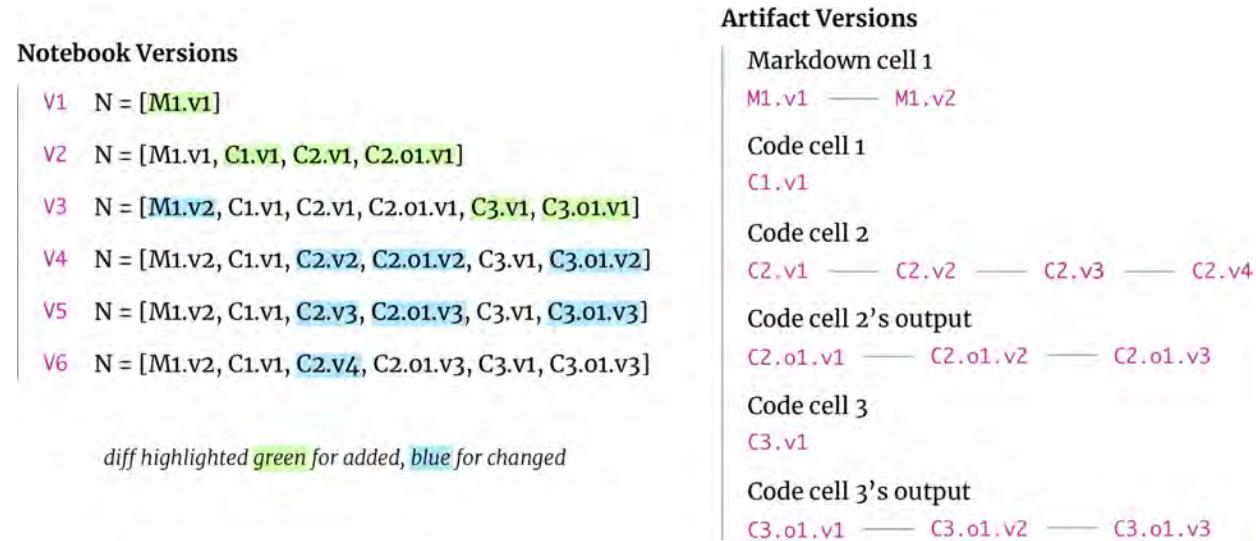


Figure 9.1 A hierarchical history model where artifacts have their own history

In this diagram, to the left we see how the notebook grows and changes over time, and to the right we see how specific artifacts change over time. There are some important relationships to note. First, all artifacts have *fewer* versions than the notebook itself, because although the user has changed the notebook 6 times (creating 6 versions), the user only changes specific cells in each version. For instance, the user only ever changes Markdown cell 1 twice, so it has two versions. Next, it's important to observe that outputs have a *many to many* version relationship. How does this happen? Take Code cell 3 as an example. C3 only ever has 1 version but has 3 different output versions. How? This happens often when a code cell contains an operation like `df.head()` which will output based on the value of an environmental variable (here `df`). Although the variable is modified in code above in code cell 2, it will result in different outputs each time C3 is run: *many* outputs to *one* version of code. Similarly, we can end up with a *one* output to *many* versions of code relationship. Here C2 has 4 different versions, but only 3 versions of output. This can occur when, for instance, the code modification between C2.v3 and C2.v4 doesn't actually change the output (e.g., the user write a code comment), such that both C2.v3 and C2.v4 share the same output C2.o1.v3.

The disadvantage of the history model in Fig 9.1 is that it adds much more complexity than simply versioning the entire notebook document v1, v2,... v6. However the advantage is that giving each artifact of interest its own history allows users to quickly look up and interact with specific components of their analysis history.

TRADE-OFFS TO AVOID COMBINATORIAL ISSUES

Another computational trade-off design decision we make is over the issue of *branching*. As shown in Variolite (Chapter 7), allowing the user to branch their history is dangerous when we've already designed a tree structure for the document itself. In typical version control, a branch allows a user to start completely different parallel timelines of history from a common point in time. So for instance, a user could backtrack to an earlier version of their notebook that they like, *branch*, and then proceed to work from that point. The issue of doing this in a hierarchical document is that it isn't clear how branching would work, and the resulting complexity of the history database could easily grow in a combinatorial and exponential fashion.

To simplify things and avoid this complexity, we do not support branching in history models used in this thesis. Instead, if the user wants to backtrack to an earlier version of their notebook and start work from there, we allow them to export that notebook version out into a new notebook file. So, notebook v73 (even though it is not the current version of the notebook) can be exported as notebook_v73, and will contain all document history up to v73 from the original document. Then this new notebook can have its own independent version history starting at that point. This is a design choice open for consideration in future work. Note that having a file named by version is eerily similar to one of the informal versioning practices reported on in Chapter 3 that we are trying to get away from. However, this design choice is a slight improvement over the informal versioning practice, since each copy of the original file will have our history data attached that makes it clear how these file copies relate to each other as versions.

HOW OFTEN TO RECORD HISTORY

Besides deciding the right granularity of *what* content to version, we need to decide the time granularity of *when or how often* we record changes. To let data scientists explore their experiment history we aim for our history store to be complete, and not missing any experiments. By trial-and-error we found versioning every time a person runs their code or changes their cell structure seems to work best. Versioning every time a person edits their notebook is *too frequent*, since in practice it means we end up storing a lot of partially-written code that doesn't compile or is incorrect.

There are normal runtime events in a Jupyter notebook that mark possible changes to the notebook. Our system listens for these events using Jupyter's APIs, and then checks for changes in the notebook:

- Save the notebook
- Load the notebook
- Run a cell
- Delete cell
- Add cell
- Move cell
- Modify cell type

If any change has occurred, the system records a new version for the specific artifact(s) that changed. Verdant also stores the time and date that version was created.

WHAT IS STORED & STORAGE CONCERNS

Shown in Figure 9.2 is a table breaking down the types of content for whose history we aim to store. Currently, our system stores all textual history for a user in a single JSON file called `<notebook name>.ipyhistory` which sits next to the user's Jupyter notebook file `<notebook name>.ipynb`. The benefit of history in a single file is that it is easily portable: a data scientist can choose to share their notebook either with or without their history file. The downside is, of course, that this isn't at all efficient. Currently, we keep the *entire* history JSON database in memory at runtime, which allows for quick reading and writing of history but takes up a lot of memory space. However, since the focus of this research is on design of user interactions with experiment history, rather than databases, we leave more efficient storage for future work. Per our hierarchical history, each version of a notebook is represented only once in the JSON file, distributed between the individual artifacts that form that version. To recreate v46 of a notebook, for instance, the system combines snippets and cell content from individual artifact versions to put back together the notebook as it existed at that time. Since most artifacts we store are plaintext user generated code or markdown, the `<notebook name>.ipyhistory` JSON file tends to remain reasonably small on disk.

Notebook	Cells	Markdown Cell
		Code Cell
		Raw Cell
	Output	Plaintext
		Images
		Tables/HTML
		Interactive
	Code Snippets	

Figure 9.2 Content in a Jupyter Notebook to consider for versioning

Issues arise with output, which are often not plaintext. In earlier versions of the history database, we treated all output as plaintext, even when they were in fact image bit code. When we piloted Verdant (Chapter 12) with real users, however, the JSON `<notebook name>.ipyhistory` file quickly became several megabytes, and since the JSON history structure stays in memory, this massively slowed down user interactions with Verdant. To solve this problem, our system now splits output by media type, and plaintext and table/html output are stored in the JSON structure with the rest of the textual artifacts. Image output is saved to file in a specially named folder we generate called `<notebook name>_output`. We argue that this is a reasonable approach, because it puts the user in control of images they generate. A data

scientist can quickly access their images from the folder, or delete them if they believe certain images are unhelpful or taking up too much space. For the purposes of our research prototypes, we ignore interactive outputs¹⁷, and leave handling of versioning interactive content for consideration in future work.

CLUSTERING & FILTERING OUT NOISE

Not every bit of code a person writes rises to the level of “an experiment”. People make syntax errors. People mess around with a line of code to figure out the right function call. People debug code that isn’t working. All of these activities are not what we would call exploratory programming or experimentation, and are not likely particularly valuable to anyone later on. Yet without adding more advanced analyses or heuristics, all coding activity looks the same to a version control system.

Collecting “junk” or “noise” or lots of versions that aren’t terribly interesting or meaningful directly hurts the usability of a history system, because the user will be faced with more work sifting through more versions to find what they are looking for. For instance, during the Jupytercon Scavenger Hunt Study (Chapter 11) we would find that participants spent far too much time scrolling through versions. Through iteration, we include several strategies to make the history we collect more meaningful.

Filtering

We attempt to *not save* results that contain programming errors by applying some filtering heuristics. Code that is not syntactically correct does not trigger any of the history save events when run, since the Jupyter API itself will not fire a “run event” if code is rejected by the programming language’s parser. Code that is syntactically correct but generates an error will display that error in output. If the output to new code is *only* of error types, we do not save either the new code or output to history. If the output contains some errors, but still outputs some not errors, we record to history as usual. Note that these heuristics still leave plenty of edge cases where erroneous code might still end up in history. If the user types some syntactically incorrect code and *does not run* it but does save the file, our heuristics will not catch that there is any error and that incorrect code will be stored as a new version. Finally, note that there may be cases in which a data worker may actually want to look at their past errors for debugging purposes. We decided that it was a worthwhile trade-off to *not support* that debugging case in favor of curating history.

Clustering

Since history save events primarily occur when a single cell or output in the notebook has changed, the resulting history tends to be many versions in which only one small thing changes. In JupyterLab, autosaves occur every 2 minutes, and are not discernable from manual user saves in Jupyter’s API. This means that history save events occur often, and often at rather arbitrary

¹⁷ Interactive outputs are HTML-type outputs in Jupyter Notebooks that can contain anything from a simple slider widget to an entire webpage. It’s worth noting that most table display outputs, due to their formatting, are also HTML-type outputs in Jupyter Notebooks. Without further heuristics and analysis, one cannot rely on Jupyter’s built-in output types to tell what kind of thing is being displayed or whether it is interactive or not.

points in the user’s work. To create fewer and more information-rich versions, we cluster some changes into the same version. If changes occur within the same few minutes (we set our threshold to 3 minutes) and are non-overlapping in that they do not affect the same cells, we combine them into the same notebook version. For instance, instead of the following log:

N V1 4:14pm user runs c14 generating a new output c14.01.v2

N V2 4:14pm user runs c15 generating a new output c15.01.v5

N V3 4:14pm user edits and run c16 generating a new cell version c16.r5

N V4 4:14pm user edits and runs c14 generating a new cell version c14.r2

We can combine events that happened in quick succession as:

N V1 4:14pm user edits c16 and runs c14,c15,16 to generate c16.r5,
c15.01.v4, c14.01.v2

N V2 4:14pm user edits and runs c14 generating a new cell version c14.r2

Note that the only change in numbering with this clustering is the notebook version number, since a single notebook version V1 is given multiple events. The final event in this log cannot be combined onto Notebook V1 and must remain in its own Notebook V2 because the two versions conflict by affecting c14. Further clustering is done on the UI level, where we experiment with visualizing similar events together, discussed later in Chapter 11.

CODE CELLS VERSUS CODE SNIPPETS AS UNITS OF MEANING

Cells are a helpful unit of work in notebooks, but flawed when it comes to our goal of tracking the provenance of code over time.

At any point in time, a cell in the notebook will very likely contain one semantically specific “job” within the user’s data analysis, much like a function has a specific “job” in a plain code file. That is how notebook documents are designed to work. However, as we can see in Figure 9.1 Notebook versions 1–6, the cell structure of a notebook evolves over time as the user adds, deletes, splits, merges, and moves around cells. Just as a programmer would rearrange and refactor code as they develop it, notebook authors move content between cells or rearrange which cells do what. Thus, to represent a notebook over time, cell-based versioning is risky, because code cells are not guaranteed to be *stable* units of meaning over time.

To illustrate this problem, consider two users: Alice and Bob. Alice tends to create a lot of small cells to test out ideas and once she has reached a good solution, she combines her finished code into one cell (see Notebook Usage Study, Chapter 4). Bob, meanwhile, prefers to iterate in a single cell for an idea over and over until he’s reached a good solution. Later, both Alice and Bob ask: how did I reach that solution again? To retrieve the history of Bob’s idea, we can simply show them v1...vn of that cell he iterated in. Alice’s history is much harder to retrieve, since that iteration was scattered across many cells, since deleted or repurposed. To answer Alice’s question we need to track the history *between cells*. In Bob’s case the history is contained at the cell-level. In Alice’s case, history happened at the level of lines of code.

To support both Bob and Alice, we can actually make our history more fine-grained and consider code snippets as artifacts themselves moving between cell containers. This design choice has significant pros and cons, discussed next.

Breaking down Code Cells into Code Snippets

To give a concrete example, consider the following code cell:

```
[223]: x = df[['id', 'bedrooms', 'bathrooms', 'sqft_living']] # features
y = np.array(df['price']) # what to predict
```

Figure 9.3 A code cell

As Python 3 code, we can break down this code into its semantically meaningful parts using python's `ast` module to convert it to an AST tree. An Abstract Syntax Tree (AST) is how the compiler of code "sees" the structure and meaning of raw code:

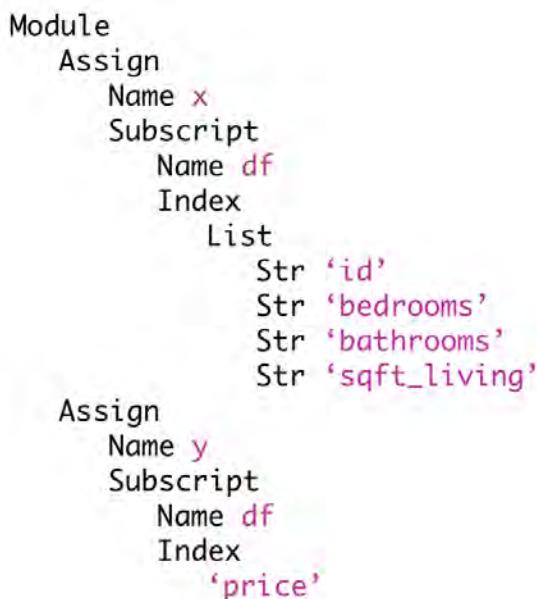


Figure 9.4 Python AST structure of the code cell from Fig. 9.3. For our purposes, “Module” is a placeholder type that just means the root of the tree.

As shown in Figure 9.1, a notebook can be described as a *list* of cell and output versions. Since our goal is to add the AST to this history representation, we say that notebook **N** is the **root node**

of an ordered tree structure describing the entire notebook. Cells and output artifacts are **child nodes**, such that:

N = [M1, C1, C2, C2.01, C3, C3.01] means **N's direct children = [M1, C1, C2, C2.01, C3, C3.01]**

We say any code cell **C** is the **parent node** for its own AST representation. So for the code shown in Figure 9.4, this code cell's direct children are the two Assign statements. The AST structure shown in Figure 9.4 is a simplified printout of what Python 3's built-in AST module will generate out-of-the-box. However, our needs for versioning code with an AST are not the same as a compiler's needs for an AST, so we do additional preprocessing. For example, notice that between the raw code in Figure 9.3 and the AST in Figure 9.4, the compiler “throws out” all the syntax tokens like `[` or `=` including all spacing and comments. Since the goal of our history database is to *preserve* the user's code in such a way that it can be recreated later, we don't want to throw out the user's comments and code style! Thus our AST includes a representation for syntax tokens and spaces so that the user's code can be recreated precisely.

Figure 9.5 shows how we adapt the AST into our history tree for the entire notebook. Since the dozens of component types in an AST are mostly language-dependent (e.g., the AST node type for Python are different from the AST node types for Julia, and even more annoyingly, AST node types for Python 2 are different from Python 3), we call *any* node of the AST a “Snippet”, and record its specific type as a property `ast_type`. For example, the Assign statement in Figure 9.4 becomes Snippet `S1.v1` in Figure 9.5. The final terminal leaves of the history tree are Snippet artifacts of `ast_type=Literal`, which contain the actual textual values of the code. For history purposes, we don't distinguish between literal and primitive types like string, double, int, etc. and treat all literal and primitive types as the same type Literal.

AST structure adds *many* additional artifacts to the versioning hierarchy. Ideally we would like to version the “meaningful” parts of this code such as the two assign statements, such that if Alice moves these assign statements across different cells, Alice can still track the history of the value of `x` and `y` over time. In Verdant 1 and 2 we version *all* elements in the AST rather than attempting to infer “meaningful” parts, which we defer to future work.

Notebook N.v33

...

Code Cell 8 C8.v5

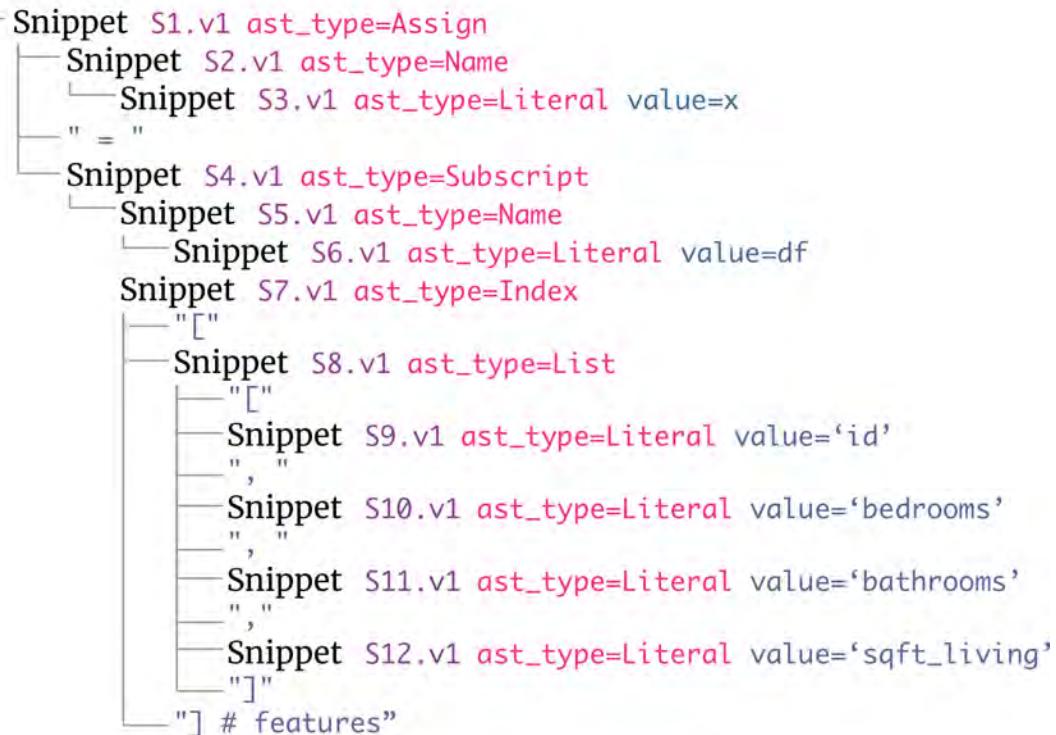


Figure 9.5 This is how the AST from Figure 9.4 translates into our hierarchical history format. Note that for a small line of code `x = df[['id', 'bedrooms', 'bathrooms', 'sqft_living']]` this adds 12 Snippets and 8 syntax literals (e.g., “[“) to the hierarchy! Unlike the compiler’s representation of code, here we represent spaces, code comments, and syntactical symbols to accurately replicate the user’s work.

Versioning at the AST level

A crucial warning to readers interested in extending this work is that versioning an AST is hard: it is both horrible to engineer and also NP-hard. The part of AST versioning that is NP-hard is the challenge of matching two trees A and B to identify the transformation that describes how A became B or else decide that A and B are completely different trees (imagine if a user pastes new code completely replacing all the contents of their code cell). Despite that dire warning, we do attempt a “good enough” AST versioning in this work with the help of heuristics, including heuristics described in prior work like variable type information, text edit position, or distance between two tree configurations [Neamtiu et al. 2005, Koschke et al. 2006].

To version at the AST level, we use the following procedure in Verdant-1 and Verdant-2. In later versions of Verdant we abandoned AST versioning for performance reasons.

1. User makes an edit. Pick the most specific possible artifact that the user edited and mark it with a ★. This marks the artifact as potentially changed.
2. When an event triggers the save of a new version (see next section about choosing save events):
 - a. **Update:** For each artifact that is marked with a ★ estimate whether it has changed using a simple textual equals:
 - i. **if not changed:** remove the ★, which will remove the artifact from further consideration.
 - ii. **if changed:**
 1. Generate the new artifact entry in the history. Process the new code through a custom parser that uses Python 3's built-in AST module to generate a new artifact tree.
 - b. **Match** the new artifact against the artifact content prior to editing. For code, this again requires program analysis using features like type, position in the AST, and string distance to estimate how the old code artifact tree should be matched to the new one. Any child-artifacts that the matching decides are either changed or new are marked with a ★.
 - c. **Commit:** Starting from the leaves of the artifact tree for the entire notebook, all artifacts marked with a ★ have a new version permanently recorded. Next, the parents of those nodes, traversing up the tree to the notebook artifact, have new versions committed to account for the new changes in their children. Finally all markers are removed.
 - d. **Save history to file.** Write the new model to the .ipyhistory file as the latest version of the user's work.

The advantage of this approach is that adding AST history allows users to retrieve the history of specific lines of code or specific parameters. Snippet history also helps users like Alice working across cells: by listening for copy-paste events in the Jupyter notebook, we can trace the provenance of individual snippets as they move from one cell to another.

The disadvantages are performance and tractability. Estimating the best match between two AST trees is computationally hard at worst, and while tractable using heuristics, we found in practice that this match step was too computationally expensive to be repeatedly executed in an interactive environment. Due to these pros and cons, some iterations of Verdant discussed below do use AST versioning, which allows for some highly promising user interaction concepts. However, as we moved into higher fidelity and deploying our history system to users, we ended up cutting AST history from Verdant due to its brittle nature and high runtime costs. Nonetheless, we believe that *some form* of code snippet versioning is a worthwhile avenue to tackle in future work, because it does add clear value to the user. In our final user study of Verdant, versioning code in units of cells alone was a major limitation noticed by users (Chapter 14). In real practice, many users are like the fictional Alice, developing code across multiple cells. This behavior is also a key part of code iteration with cells that we noted in the Notebook Usage Study (Chapter 4). Thus we believe it is worthwhile to revisit the code snippet problem, but also highly advise those who embark on this issue: *what would be good enough from the user's perspective?* The solution to performative code snippet versioning is, I believe personally, to reframe the problem in such a way that avoids AST matching all together. Instead of trying to

version at such a fine grain as AST nodes, the more pressing problem of versioning code as it migrates between cells (Chapter 14) could be approached without ASTs at all.

CHAPTER CONCLUSIONS

In this chapter we've described our approach for the history model used in the remainder of this dissertation. The key questions outlined in this chapter, such as how often to version, what granularity to version at, and how to store history for fast retrieval are design questions that have multiple viable answers and implementations. For a reader interested in building their own history model for fine-grained relational history of artifacts, we encourage you to use this chapter as a worked example for how we addressed each design question for Verdant, but these design questions are worth revisiting for a new context. For instance, to adapt Verdant-style history into something that works directly with Git requires re-considering the time units at which history will be collected, and how a relational history structure will fit into Git's own idioms. To translate Verdant-style history into a normal Python script file instead of a notebook file requires reconsidering what the units of an “artifact” should be in the absence of a cell structure. One possibility is line-level or statement-level artifacts, in a similar fashion to Git “blame” functionality. Finally, an important theme to take away from this chapter is that the structure of history data, and all of these design choices for data collection, are first and foremost about creating a user experience. In the case of history, the version data is the primary thing that users will directly interact with, so getting the structure of versioning right cannot be done in isolation as a “back-end” systems effort. Instead we encourage those implementing a system like this to test early and often how the data being collected will appear and be used by practitioners.



Chapter 10: Interactive Versioning within a Notebook

Featuring Verdant-1

Research in this chapter was done in collaboration with Brad A. Myers¹⁸

INTRODUCTION

This chapter combines our earlier work on Variolite (Chapter 7) and Rose Quartz (Chapter 8) with our Notebook Usage Study from Chapter 4. In Verdant-1 we move our design goal to capturing a full picture of a user’s experiment *within a notebook*. This new scoping of our design goal is informed by our work in Rose Quartz. Unlike the difficulties we faced collecting workflows between multiple tools with Rose Quartz, a Jupyter notebook serves as a sandboxed environment where we can readily collect a semi-complete picture of a data worker’s experiment workflow. We say “semi-complete” because a notebook encapsulates a user’s notes, code, and output but does not hold the data itself that the user is working on. While we leave the limitation of data history for future work, scoping to a notebook environment allows us to start experimenting with far more types of designs driven by real experiment data. In Verdant-1, we begin with interface design ideas very similar to what we had in Variolite. Informed by the limitations of Variolite’s history model, we use the new history model discussed in the prior Chapter 9. A complete and automated history of the user’s notebook with that model allows us to refresh snippet versioning ideas from Variolite with new flexibility. Having the history of *everything* opens up the possibility of giving the user history of *anything* at a moment’s notice.

DESIGN GOAL: CREATE A BETTER HISTORY EXPERIENCE FOR EXPERIMENTING IN NOTEBOOKS

With a complete history of a data scientist’s work in a notebook (Chapter 9), we now have all the data we need to prototype user interactions. Collecting the appropriate versioning is just one end of the problem. Without carefully designed interactions, we know that a pile of log data is not usable. A key problem is the sheer number of versions. 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

¹⁸ This chapter is based in part on the conference paper: Mary Beth Kery, and Brad A. Myers. "Interactions for untangling messy history in a computational notebook." In 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 147–155. IEEE, 2018.

overwhelming for a human to interpret [Pimentel et al. 2015]. 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 [Ragavan et al. 2016]. 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 [Codoban et al., 2015]. In our own study (Chapter 3) we saw that even when data scientists keep history of their exploratory code, e.g., v1...vn of a file, they struggle to recall which versions contain what. We hypothesize that making experiment history fast and easy to browse, search and interrogate is the key to making those past versions useful to data scientists.

In this prototype, we explore new interactions for providing easy-to-use history support for data scientists in their day-to-day tasks. We explore the design space of fast and lightweight interactions for tasks such as:

- The user wants to quickly get to the history of an artifact, for instance to backtrack.
- The user wants to compare between versions of different artifacts including code, tables, and images, which benefit from different diffing techniques.
- The user wants to reproduce a specific version of an artifact

PAPER PROTOTYPING

Prototyping was conducted at the HCII at Carnegie Mellon University with the help of Brad Myers who contributed to the design.

In early sketches and mockups we were interested in visualizing history as a dimension of code. Since the notebook document contains cells ordered top to bottom, we experimented with displaying history right to left as a horizontal ribbon of versions overlaid on top of the code (Figure 10.1).

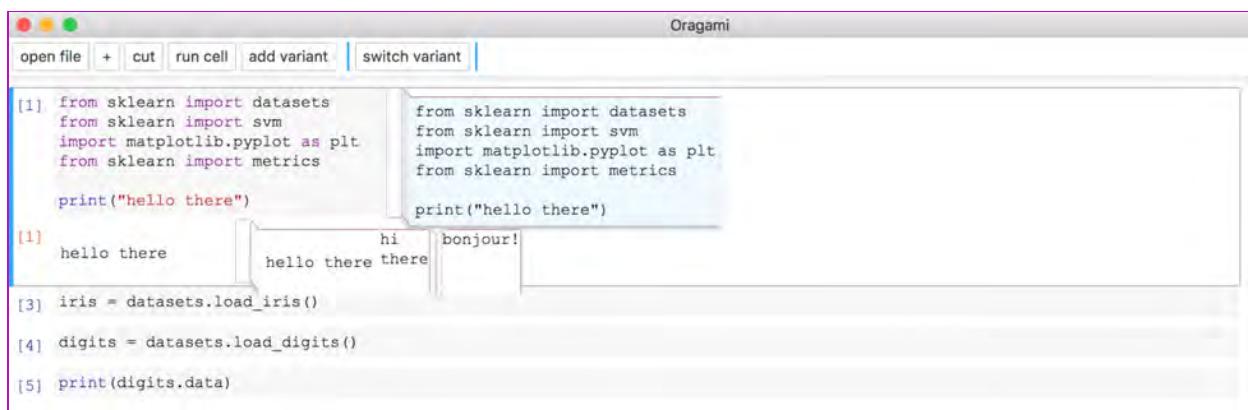


Figure 10.1 Sketch of horizontal ribbon concept



Figure 10.2 Sketch of origami folds concept

We also experimented with showing collapsed/hidden versions as ‘folds’ in the horizontal history ribbon. This early paper prototype concept was nicknamed “Origami” for the paper visual metaphor, but was eventually discarded for being too decorative (Figure 10.2).

SYSTEM: VERDANT-1

To test our designs, we developed a prototype tool called Verdant-1¹⁹ (from the meaning “an abundance of growing plants” [Wiktionary 2018]) as an extension for Jupyter notebooks. Verdant-1 is built as an Electron app that runs a Jupyter notebook, and is implemented in HTML/CSS and Node.js.

Note that Verdant-1 implements the history model discussed in Chapter 9. The underlying naming scheme used in the history database from Chapter is *not displayed* to the user in this prototype, but is still the same in Verdant-1’s back-end.

Although a notebook may contain many code, output, and markdown cells, prior work suggests that data scientists work on only a small region of cells at a time for a particular exploration (Chapter 4). First, we show how Verdant-1 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.

Ambient Version Indicators

Following tried and tested usability conventions of other tools that support investigating properties of code, such as 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-1 takes the approach shown in Figure 10.3, where a number in the right margin of the notebook cell indicates how many versions exist for a given artifact. 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

¹⁹ It was just named “Verdant” in the original paper [Kery & Myers 2018], but renamed in this thesis to be Verdant-1 to distinguish it from subsequent versions.

about. Finally, if the programmer clicks on the indicator, this will open the default active view, the ribbon display (Fig 10.3, 4) with buttons for reading and working with the versions of that artifact, as described next.

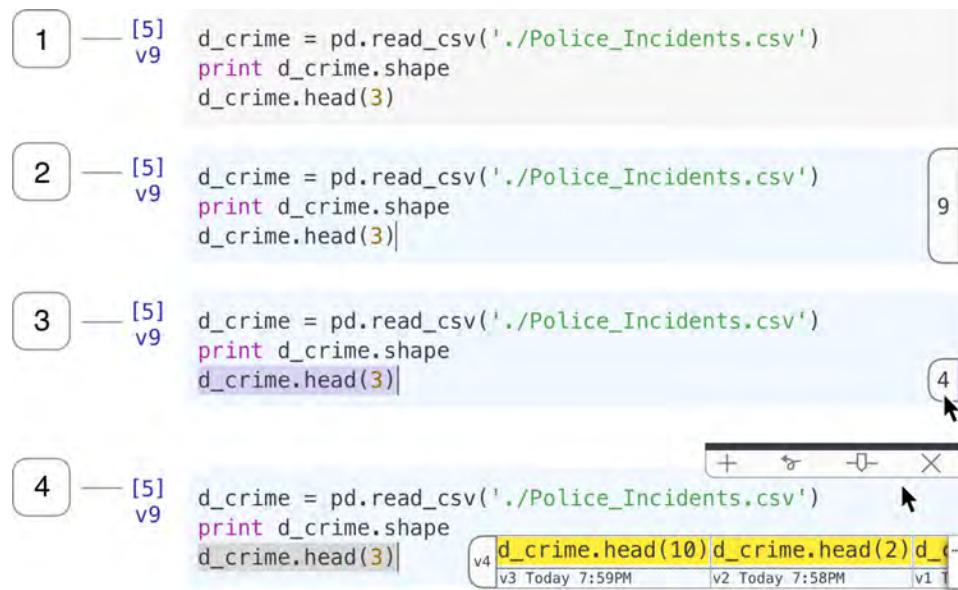
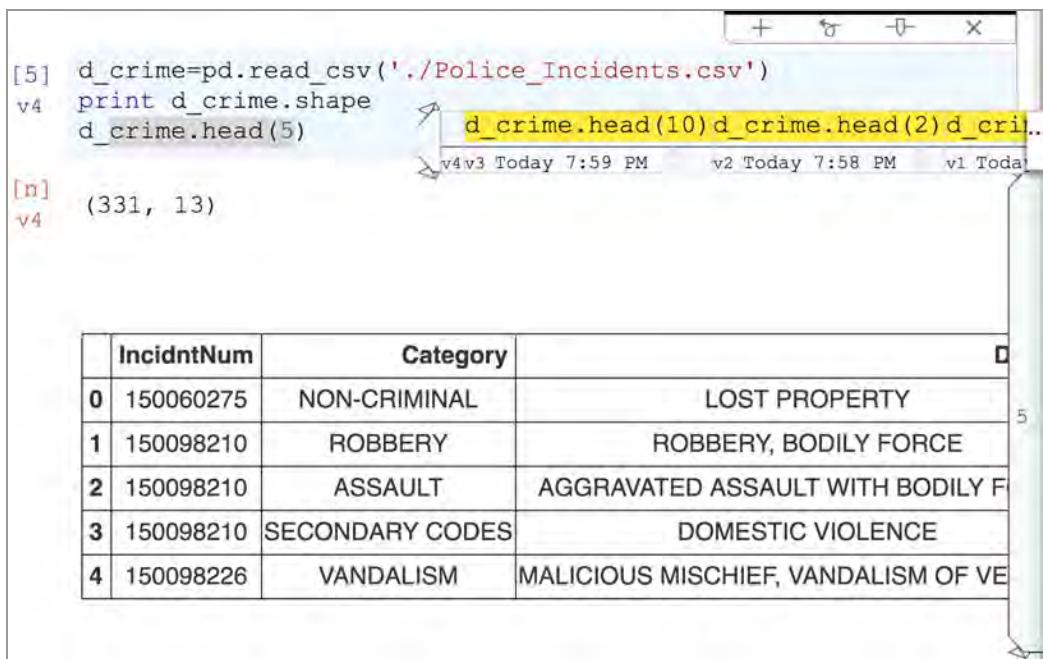


Figure 10.3 In (1) the user selects code of interest, here a code cell. Instead of explicitly creating a variant box, in (2) a passive indicator to the right of the selection indicates how many specific versions exist for that code. In (3), the user selects a single line of code with their cursor, and the indicator changes to show that there are just 4 versions of that code, despite 9 versions of the entire code cell. If the user clicks on the indicator (3), a variant box and header appears (4), that lists all versions of that code. Besides code, the variant box interaction also works for markdown and output cells.

Navigating Versions

The “ribbon display” shown in Figure 10.4 is the default way Verdant-1 shows all versions for an artifact, lined up side by side to the right of the original artifact. We struggled to fit versions in the kind of standard hovering pop-up used by code linters or autocomplete to supply information inline with code. Versioning data takes up a lot of screen space since it consists of a long ordered list of information and is continuously updated as the data scientist runs their code. So in the ribbon visualization, because code 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 10.4), 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 scrolling, 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. 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.



The screenshot shows a Jupyter-style notebook interface with two cells. The top cell contains Python code for reading a CSV file and printing its shape and head. A yellow ribbon highlights the third line of code, 'd_crime.head(10)'. Below the code cell is an output cell showing the result: a table with 331 rows and 13 columns. To the right of the output cell, a vertical margin indicator shows the number '5' with a downward arrow, indicating there are five previous versions of the output.

```
[5] d_crime=pd.read_csv('./Police_Incidents.csv')
v4 print d_crime.shape
d_crime.head(5)
d_crime.head(10)d_crime.head(2)d_crime...
[n]
v4 (331, 13)
```

	IncidentNum	Category	
0	150060275	NON-CRIMINAL	LOST PROPERTY
1	150098210	ROBBERY	ROBBERY, BODILY FORCE
2	150098210	ASSAULT	AGGRAVATED ASSAULT WITH BODILY F
3	150098210	SECONDARY CODES	DOMESTIC VIOLENCE
4	150098226	VANDALISM	MALICIOUS MISCHIEF, VANDALISM OF VE

Figure 10.4 Verdant-1 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.

Comparing Versions

Among many versions, it is important for a data scientist to quickly pick out what is important about that version out of lots of redundant content. In Verdant-1, a diff is shown in the ribbon and timeline views by highlighting different parts of a prior version in bright yellow (Figure 10.4). For code, Verdant-1 runs a line-oriented textual diff algorithm consistent with Git, and for artifacts like tables that are rendered through HTML, Verdant-1 runs a textual diff on the HTML versions and then highlights the differing HTML elements. Differing can help a user narrow down what they are looking for. 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 by highlighting just that line's code with her cursor to show only versions in which that line changed, hiding all other versions that are not relevant to that change.

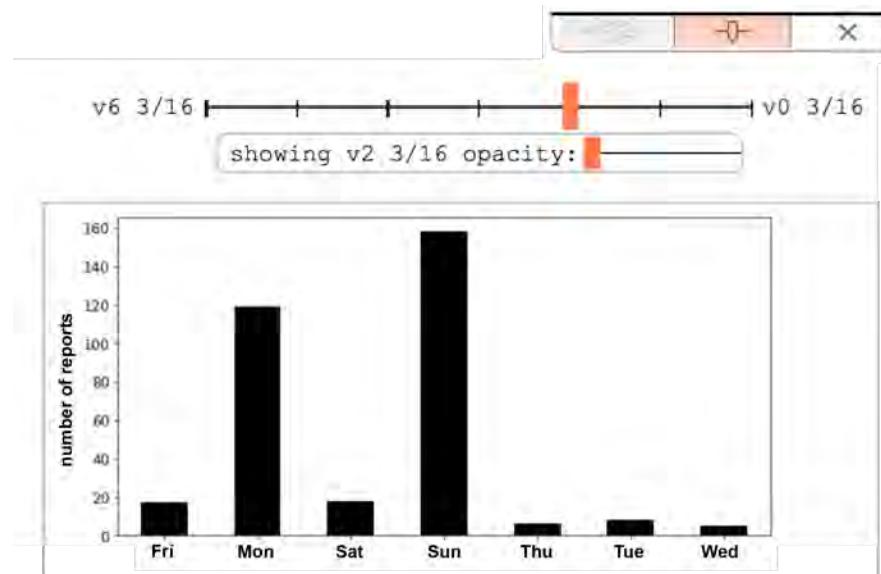


Figure 10.5. Timeline view. By dragging the top orange bar side to side the user can change the version shown. By dragging the lower orange bar, the user can set the opacity of the historical version they are viewing, in order to see it overlaid on top of the currently active output version.

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 [Gleicher et al. 2011] 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 10.5), 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-1 again relies on advice from visualization research [Gleicher et al. 2011] and uses opacity so the user can change the opacity of a version they are looking at to see it overlaid 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:

- What is the difference between the active version and a given prior version?
- What changed in version N from the version immediately prior?
- 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-1, 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).

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 (Chapter 4). 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, 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-1 also represents all versions in a list side pane (Figure 10.6).

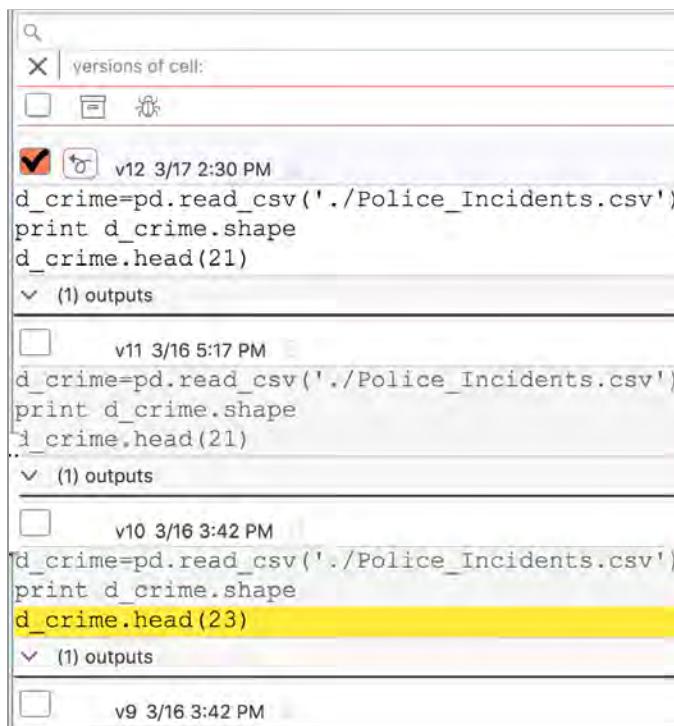


Figure 10.6. In the list view, the user can select one or more versions to act upon. With the search bar, the user can filter versions using keywords or dates.

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 [Pimentel et al. 2015].

Replay older versions

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 (Chapter 4). Without history, reproducing results is commonly a tedious manual process, where the data scientist needs to find or re-create the original code from memory (Chapter 3).

To replay any older version of an artifact in the notebook, a user in Verdant-1 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-1 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-1 provides a feature called “Recipes”, described next.

	IncidentNum	Category	Description	Date
0	150060275	NON-CRIMINAL	LOST PROPERTY	Mon
1	150098210	ROBBERY	ROBBERY, BODILY FORCE	Sun

```

step # 0 v0 Today 7:15 PM
import sys,os
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

step # 1 v9 Today 7:15 PM
d_crime=pd.read_csv('./Police_Incidents.csv')
print d_crime.shape

```

Figure 10.7. 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.

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 10.7). Verdant-1 uses the chain of dependency links that it has calculated from the output to produce a recipe visualization, shown in Figure 10.7. The “recipe” appears in the side list pane as an ordered list of versions labeled “step 0” 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 entirely in yellow. If a matching code cell already exists in the notebook and perfectly matches the active version, it will be shown entirely in 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 do not address the issue of versioning datasets in the scope of this thesis.

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 (Chapter 3) [Patel et al. 2008]. Thus, especially when collaborating with others, it can be important for a data scientist to communicate which paths failed (Chapter 4) [Codoban et al., 2015], and how they got to a certain solution. “Which path failed” requires a kind of storytelling that it is 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 [Codoban et al., 2015]. Under what circumstances would data scientists be motivated to label trustworthiness and relevancy of their code? To experiment with interactions for this purpose, Verdant-1 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. 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 think they would actively manually tag versions like this.

VERDANT-1 USABILITY PILOT

Verdant-1 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 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-1 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-1 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 notebooks²⁰ that are curated for quality by the Jupyter project [Jupyter Project 2018]. 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 Francisco²¹. Since currently detailed history data is not available for notebooks, we edited and ran different variations of the San Francisco 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

²⁰ <https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks>

²¹ lmart, “SF GIS CRIME,” GitHub. <https://github.com/lmart999/GIS>.

30 to 50 minutes and participants were compensated \$20 for their time. Participants will be referred to as D01 to D05.

All participants were able to successfully complete the tasks, suggesting at least a basic level of usability. Among even this small sample, we were surprised by the diverse use cases participants expressed that they had for the tool. D01 and D05 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. D02 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. D03 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. D03 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, D03 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 D04 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. D02, D03, and D04 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.

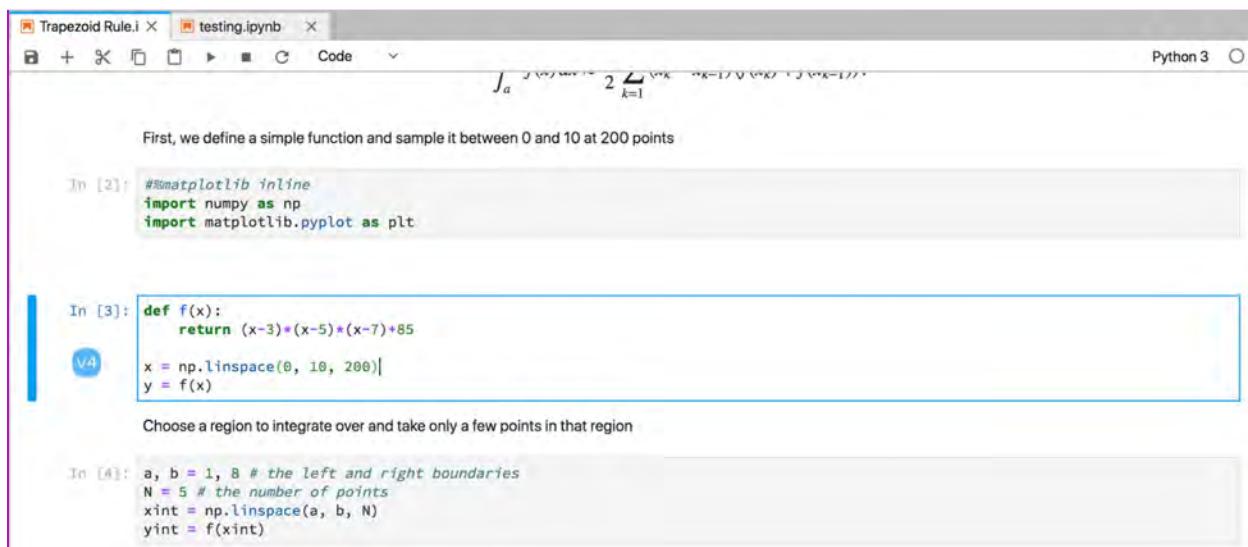
In this initial study, a participant's imagined use case affected which features of Verdant-1 they cared about most. When probed about the use of bookmarking, D02 felt strongly that bookmarks would be useful for their use case of switching among a few different alternative versions, however the other participants who had more history-based use cases 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. D01 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. D02 said they would likely mark versions as buggy, but would be wary of using the archive button to hide older or unsuccessful content. D02 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. D05 said that although they imagined themselves tracing an output's dependency rarely, this feature was extremely valuable to them when needed, since currently when D05 must recreate output with today's tools, this was a tedious and error-prone manual process of trying to recode 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-1, 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. D03, 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 Verdant-1) 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 preferred the ribbon's dropdown menu to select versions.

DESIGN ITERATION

Following the usability study, we did another round of paper prototyping. In the following several iterations we focused on sketching a more polished look that could integrate into Jupyter Lab, as opposed to the Jupyter Notebook clone Verdant-1 is implemented as. We also focussed on redesigning the UI layout to get rid of the ribbon visualization, since the horizontal ribbon is problematic to scroll and did not scale particularly well. These designs were not implemented, due to our shifting design focus, discussed next.



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Shows "Trapezoid Rule.i" and "testing.ipynb".
- Toolbar:** Includes icons for file operations, code execution, and help.
- Header:** Shows "Code" and "Python 3".
- Plot Area:** Displays a trapezoidal approximation of a function from a to b, divided into 2 segments with points labeled k=1 and k=2.
- In [2]:** A code cell containing:


```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```
- In [3]:** A code cell containing:


```
v4
def f(x):
    return (x-3)*(x-5)*(x-7)+85
x = np.linspace(0, 10, 200)
y = f(x)
```
- In [4]:** A code cell containing:


```
a, b = 1, 8 # the left and right boundaries
N = 5 # the number of points
xint = np.linspace(a, b, N)
yint = f(xint)
```
- Text:** "First, we define a simple function and sample it between 0 and 10 at 200 points"
- Instructions:** "Choose a region to integrate over and take only a few points in that region"

Figure 10.8. We moved ambient indicator `v4` to the cell margin where the runtime kernel number `In[3]` is shown. This fits the visual style of the Jupyter notebook layout a bit better than the original indicators in Figure 10.3.

In [3]:

```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return (x-3)*(x-5)+85
    return (x-3)*(x-5)*(x-7)+85
    return (x-3)*(x-5)
    return (x-3)

Choose a region to integrate over and take only a few points in that region
```

In [4]:

```
a, b = 1, 8 # the left and right boundaries
N = 5 # the number of points
xint = np.linspace(a, b, N)
yint = f(xint)
```

Figure 10.9. Here we again experimented with showing versions as a pop-up overlay similar to a code completion pop-up. However, this idea was discarded because it scales poorly with large chunks of code or artifacts with many versions.

In [2]:

```
#%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

v4

In [3]:

```
def f(x):
    return (x-3)*(x-5)*(x-7)+85
    return (x-3)*(x-5)+85
    x = np.linspace(0, 10, 200)
    y = f(x)

Choose a region to integrate over and take only a few points in that region
```

v2 MAY 29TH 8PM

In [4]:

```
a, b = 1, 8 # the left and right boundaries
N = 5 # the number of points
xint = np.linspace(a, b, N)
yint = f(xint)
```

Figure 10.10. To replace the ribbon display (Figure 10.3), here we show versions in a side-by-side view with the user’s current code to the left and an older version to the right. The downside of this display is that the user can only see two versions at a time, and the side-by-side layout does not work so well for large chunks of code/output/markdown when each version requires a lot of horizontal screen space.

The screenshot shows a Jupyter Notebook interface with two code cells. The top cell (In [2]) contains a comment and imports for matplotlib and numpy. The bottom cell (In [3]) contains a function definition and its execution. A blue vertical bar highlights the function definition in both cells. The cell header for In [3] includes a 'DIFF' button. The bottom cell (In [4]) contains code to generate points for integration. The interface has tabs for 'Code', 'RECIPE', and 'OVERLAY'. The title bar shows 'Trapezoid Rule.i' and 'testing.ipynb'.

```
In [2]: #%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

In [3]: def f(x):
    return (x-3)*(x-5)*(x-7)+85
x = np.linspace(0, 10, 200)
y = f(x)

In [4]: a, b = 1, 8 # the left and right boundaries
N = 5 # the number of points
xint = np.linspace(a, b, N)
yint = f(xint)
```

Figure 10.11 The same side-by-side concept as Figure 10.10 but here we add diff highlighting to the code and add in all of Verdant-1's history features into the cell header overlay.

This screenshot shows a Jupyter Notebook interface with two code cells. The top cell (In [48]) contains code to print JSON output. The bottom cell (In [49]) contains a large amount of textual output from the 'main' function, with some parts highlighted in red. The cell header for In [49] includes a 'DIFF' button. The interface has tabs for 'Code', 'RECIPE', and 'OVERLAY'. The title bar shows 'Trapezoid Rule X' and 'testing.ipynb'.

```
In [48]: nodey['content'].pop() #remove end marker
print (json.dumps(nodey, indent=2))

In [49]: text = """
def f(x):
    return (x-3)*(x-5)*(x-7)+85

x = np.linspace(0, 10, 200)
y = f(x)"""

main(text)
#print(json.dumps(main(l, tree), indent=2))

In [50]: MAY 14 7:34PM RUN #22 ← → RECIPE
Treating Module

Treating FunctionDef
got before_tokens [{"start": {"line": 2, "ch": 0}, "end": {"line": 2, "ch": 3}, "type": "NAME", "literal": "def"}, {"start": {"line": 2, "ch": 4}, "end": {"line": 2, "ch": 5}, "type": "NAME", "literal": "f"}, {"start": {"line": 2, "ch": 5}, "end": {"line": 2, "ch": 6}, "type": "NL", "literal": "\n"}]
bounty: []
```

Figure 10.12 Here we show the same ambient version indicators as Figure 10.8 and the same in-context header display as Figure 10.11, except this time for output. Shown is a diff for textual output where the diff is inline, and the user can pick which kind of output diff they want using the dropdown menu next to the button labeled 'diff'.

LIMITATIONS OF SCALE

Our design explorations before and after the initial usability test of Verdant-1 all were dogged by issues of scale. What do we do when each version of an artifact requires a lot of screen space? Lots of possible designs look visually good with 2-3 versions of an artifact, but what happens when each artifact has 30 versions on screen? Scale problems affect the visuals of our UI design, but also user interactions. Verdant-1's list pane, shown in Figure 10.6, works reasonably well when there are not *too many* versions, however we know from prior work and our own experiments with history modeling that a user can easily accrue hundreds of versions of their notebook within a few hours of working on it.

Due to these limitations, we decided to pivot to tackling *interacting with history at scale* as our next step of research, discussed in Chapter 11.

CHAPTER CONCLUSIONS

Verdant-1 was well received by both our pilot users and the research community. Even as we move on to focusing on history at scale and different research problems in the following systems of this dissertation, parts of Verdant-1 remain exciting. With automatic history collection going on silently in the background as a practitioner works, in-line version access in Verdant-1 comes as a somewhat magical experience: click on anything in the notebook, it's specific history just appears. That's it. The simplicity of having all of that information available at a click is exciting in that all that history data would never have been available in ordinary data work practice (Chapter 3), but simply appears without any effort at all in Verdant-1. While in-line versioning on click does not scale to browsing large amounts of history, it is a feature that has been consistently popular and requested by participants trying out our history tools in the rest of this dissertation.

Next, I will try to tie together some of the major design takeaways from Verdant-1:

- + **Inline access of the history of any artifact is a wonderful thing:** We found that the ability to click on specific content and see its specific history is consistently popular with users of our prototypes, in Verdant-1 but also in subsequent studies of Verdant 2 and 4. The appeal is that users are able to see the history of what content is right in front of them without having to search or skim through a bigger pile of history logs. This interaction is also similar to Git blame, although our history tools keep much more detailed history.
- + **Help users reproduce results as a “recipe”:** In our pilot usability study and reception by the research community, the metaphor of reproducing output as a “recipe” was easy for users to understand. In Verdant-1, the recipe UI shows just the code steps needed to recreate an output, but not the “ingredients” such as which data file needs to be run. To fully realize the recipe UI metaphor, more data would need to be collected at runtime, similar to the program slicing method used by Head et al. [Head et al., 2019]
- **Inline history visualization makes it difficult to see how versions of different artifacts relate:** Inline history access is most beneficial for seeing a *single* artifact's history. We found in prototyping that showing all the versions of one artifact A and elsewhere

showing all versions of a different artifact B allows users to examine A and B individually but tells the users very little about how the histories of A and B relate. With limited screen space in an inline history visualization, and since A and B can be quite a distance from each other in document placement, different UI approaches are needed to make history relationships clear.

- **Inline history visualization does not scale well:** Just like we found that tab UI in Variolite works best with just 2-4 versions, we found that inline history visualization works best for users when the user is comparing just 2 versions of the same artifact side by side. In later designs we incorporate different selection dropdowns so the user can pick which 2 versions to compare, but due to screen space limits, usually just 2 versions can be seen on screen without scrolling, regardless of whether we put versions side-by-side horizontally or vertically.



Chapter 11: Recovering Experiment History Facts at Scale

Featuring Verdant-2

Research done in collaboration with Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers²²

INTRODUCTION

Again in this chapter, we shift to look at a different part of the design space. Verdant-1 in the previous Chapter 10, like Variolite before it (Chapter 7), was focused on helping users access the history of what they can see in front of them in their current document. Both Verdant-1 and Variolite featured an interaction where a user could simply click on content to access its history. For cells or code that no longer exist in the current document, both Verdant-1 and Variolite deferred all of this other history to a side pane UI that simply lists all other history in chronological order.

By the end of our testing in Verdant-1, we became confident that our designs really do address the needs we identified back in our Exploratory Programming Study in Chapter 3, to let practitioners quickly access versions of any specific content in their document. However, we had, to this point, done very little to address the issue of scale. In our Exploratory Programming Study, our Notebook Usage Study (Chapter 4), and in Rose Quartz (Chapter 8) we repeatedly found that *having history* isn't the same as it being usable or useful. Now in Verdant-2 we decided to tackle this issue, and turn to focusing our design efforts on making *large amounts* of history data *useful* to practitioners.

DESIGN GOAL: SUPPORT FAST & EASY HISTORY RETRIEVAL AT SCALE

If a data scientist wants to answer a concrete question from their prior work, such as “*why did I discard this data feature from my model?*”, they will need that:

1. **The history is sufficiently complete:** the experiments that led to each particular choice must have been recorded in the first place in enough detail to understand or replicate.

²² This chapter is based in part on the conference paper: Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. "Towards effective foraging by data scientists to find past analysis choices." In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–13. 2019.

2. The cost of tracking down an answer is not prohibitive. History is reasonably easy to search, compare, and understand.

Verdant-1 reasonably achieved (1) by automatically capturing checkpoints of a data scientist’s notebook at regular intervals, such as every time a user runs their code or saves their notebook. However, as we discovered in our initial user study, some of the user interactions in Verdant-1 don’t scale to dozens or hundreds of versions.

The cost of retrieving useful information out of long history logs typically *is prohibitive*, but we have reason to believe that solving this problem could greatly benefit data scientists. If data scientists and others could effectively interrogate the history of an analysis or model, it would make the underlying choices an author made more transparent: *why is the model this way? Why not this way?* Analysis and model interpretability, transparency, and robustness are currently of great interest to society as a way of verifying the quality of the models and analyses that impact the world.

Prior research provides some hints for foraging in history. 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 [Codoban et al., 2015][Ragavan et al. 2016]. Srinivasa Ragavan et al. have modeled how programmers navigate through prior versions using Information Foraging Theory (IFT) [Ragavan et al. 2016] in which a programmer searches for the information by following clues called “scents”. Information foraging theory (IFT), developed by Pirolli and Card [Pirolli 2007], stems from the biological science concept of optimal foraging theory as applied to how humans hunt for information. IFT includes certain constructs adopted from optimal foraging theory: predators correspond to humans who are hunting for information, their prey. They perform these hunts in parts of the UI, called patches. In the context of foraging in software engineering, the programmer is the predator, the patch is an artifact of the environment which can vary from a single line of code to a generated output or a list of search results, and the piece of information that the programmer is looking for is the prey. A cue is the aspect of something on the user’s screen that suggests a particular place that they should look next.

IFT has been applied to source code foraging in a variety of domains including requirements tracing, debugging, integrated development environment (IDE) design, and code maintenance [Fleming et al. 2013, Lawrance et al. 2007, Lawrance et al. 2008, Perez & Abreu 2014, Piorkowski et al. 2015]. The design of our tool builds upon this work by taking into account design implications for how programmers forage for information [Guo Dissertation 2012, Lawrance et al. 2007, Perez & Abreu 2014] by providing specific foraging cues such as dates, previews, and diff highlighting.

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.

DESIGN PROCESS

Unlike our prior prototypes, Verdant-2 was prototyped during a summer internship at Bloomberg L.P. where Bonnie E. John, a researcher and UX designer, and Patrick O’Flaherty, an interaction designer with a visual design background, both contributed greatly to the design. Brad A. Myers gave design input as well. During the design process we received some feedback from the UX team at Bloomberg as well as from data analysts and machine learning experts at Bloomberg.

Usage Scenarios

Given the breadth of data science tasks, we first analyzed available data on specific questions data scientists have articulated that they want to understand from their history (Query Design Exercise, Chapter 4). We used these data to map out use cases to guide our design:

- (1) A data scientist is working alone with their final results as the deliverable. Over a long period of work, they use history as a memory aid to check their intermediary results. (Verdant-1 usability pilot participant, Chapter 10)
- (2) A data scientist is communicating their in-progress experimentation to a colleague. For instance, an analyst is using history to justify a model to her boss. (Exploratory Programming Study participant, Chapter 3)
- (3) History is sent along with a data science notebook for process transparency. For instance, a professor can use history to understand how a student got to a specific result on an assignment (“show your work”). (Verdant-1 usability pilot participant, Chapter 10)

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

Sketches

In our initial design mockups, we focused on the problem of making a long list of versions of the entire notebook readable such that a data scientist might quickly skim over the list and spot the version they’re looking for.

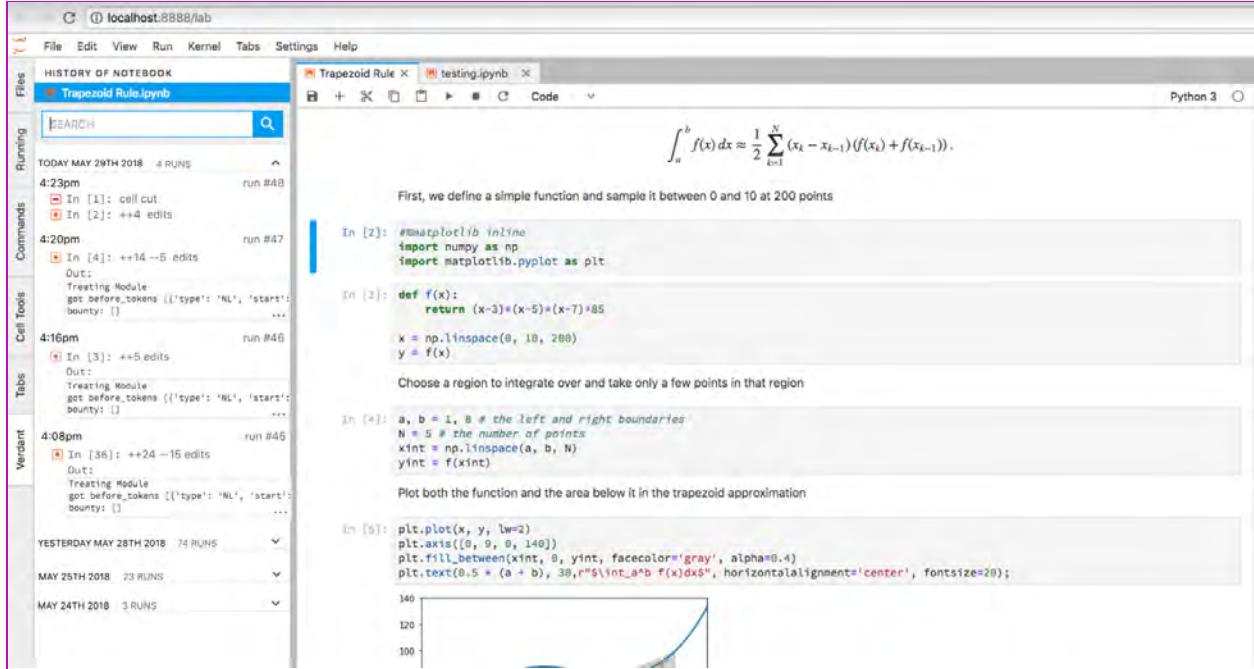


Figure 11.1. In this concept for a history side pane (left of the notebook) notebook versions are organized in an “activity feed” by date and time. We experimented with visualizing what changed in each version. For instance the block with the minus sign in run #48 in the first entry at 4:23pm indicates that the cell was removed. The block with the circle indicates code was changed, with a label $++4$ edits indicating that 4 characters were added to the code.

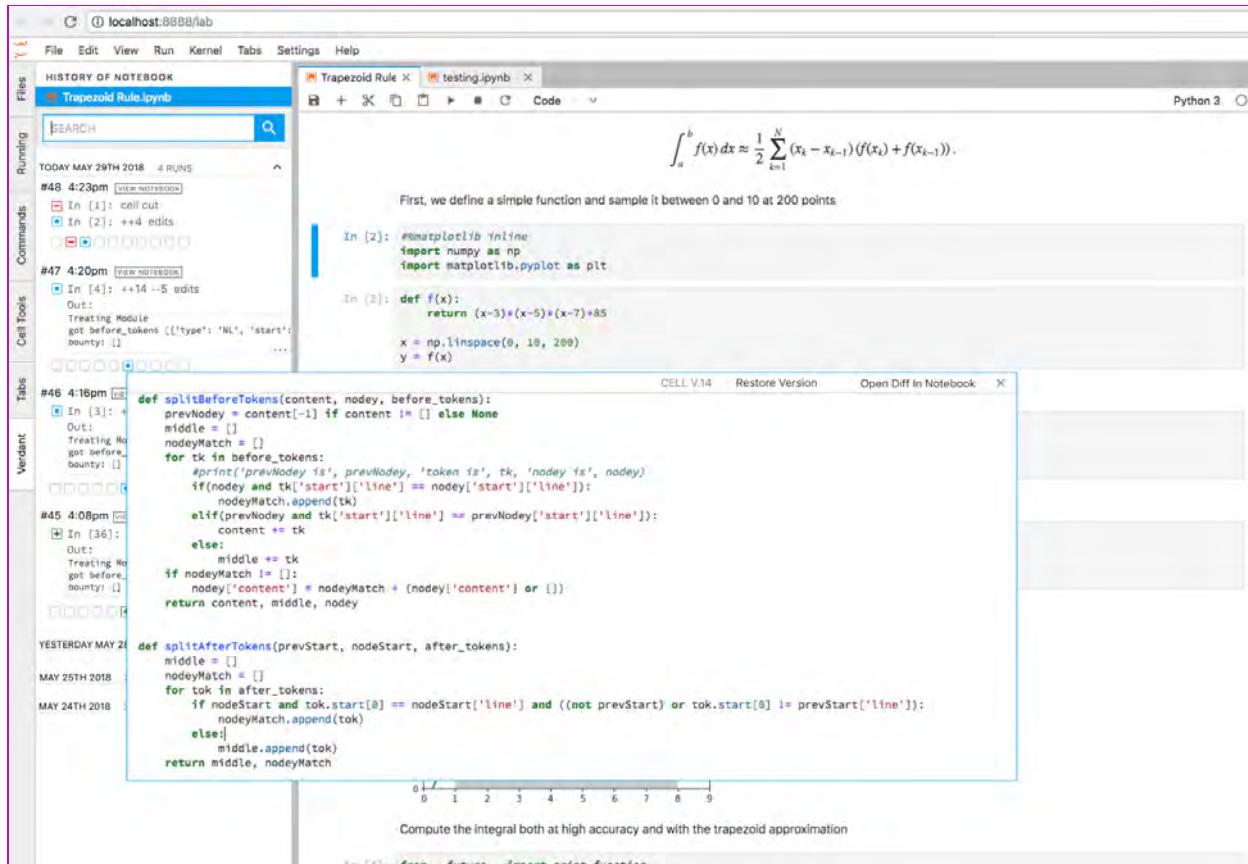


Figure 11.2 Here we first introduce a notebook “minimap” as a way of summarizing at-a-glance what happened in the notebook at a particular version. In version #48 at 4:23 the horizontal row of 10 blocks represents the 10 cells in this notebook at that time. The fill color/icon of each block indicates if that cell was changed in this version. In #48 the red block was a cell deleted and the blue block was a cell edited. In #47 we show the user hovering their cursor over the blue block in the minimap. An overlay popup appears that previews what that particular cell looked like in version #47.

The screenshot shows a Jupyter Notebook interface. On the left, there is a vertical sidebar with sections for 'File', 'Edit', 'View', 'Run', 'Kernel', 'Tabs', 'Settings', and 'Help'. Below this is a 'HISTORY OF NOTEBOOK' section with a search bar. The history lists several runs from May 29th, 2018, including #48 at 4:23pm, #47 at 4:20pm, #46 at 4:16pm, #45 at 4:08pm, #44 at 4:04pm, and #43 at 4:02pm. To the right of the sidebar is the main notebook area. The current tab is 'Trapezoid Rule.ipynb'. The code in the cell shows the implementation of the trapezoidal rule for numerical integration. The output cell displays a mathematical formula: $\int_a^b f(x) dx \approx \frac{1}{2} \sum_{k=1}^N (x_k - x_{k-1}) (f(x_k) + f(x_{k-1}))$. Below the formula, the text reads: 'First, we define a simple function and sample it between 0 and 10 at 200 points'. The code then defines a function `f(x)`, imports numpy and matplotlib.pyplot, generates 200 points between 0 and 10, and calculates the integral using the trapezoidal rule.

Figure 11.3 In this progression of the “minimap” visualization, we remove the original text summaries of each version (Figure 11.1) in order to fit more notebook versions on the screen. We also replace the horizontal row of cubes from Figure 11.2 with a more condensed minimap styled as underlines or blocks if changed.

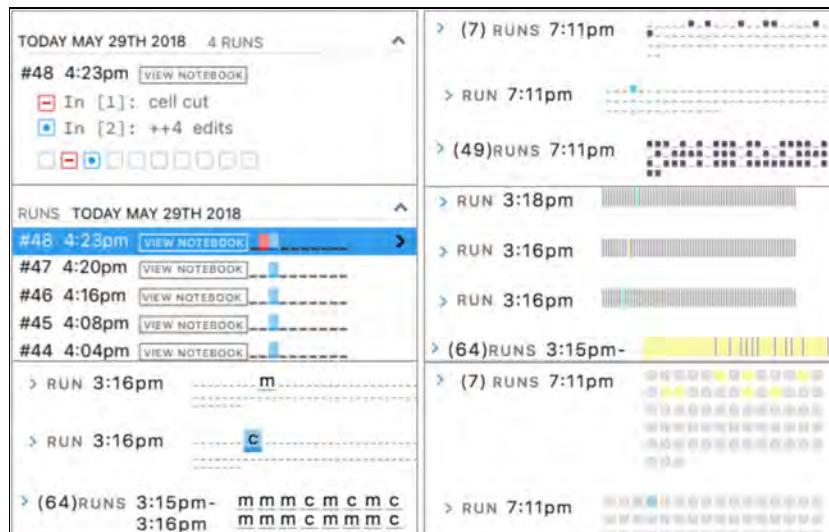


Figure 11.4 We explored a number of different minimap styles. Our requirements for the minimap were to: 1) fit as much information to summarize the notebook version as possible, 2) scale up to a notebook containing 60 cells (this number is from our study results in Chapter 4 as a “max” reasonable size for a notebook), and 3) make minimaps between versions align such that the user can visually compare between minimaps of different versions. Ultimately the scale issue was the most challenging, leading us to choose a minimalist “tic” design (Figure 11.7) similar to a sparkline visualization.

The screenshot shows a code editor interface with a Python script. A tooltip or on-hover visualization is displayed over a line of code, specifically over the character ']' in the expression `target = bounty[-1] if len(bounty) > 0 else None`. The tooltip contains a list of commit history entries from a version control system, likely Git, showing changes to the file. The commits are as follows:

- v26 ++d--14 MAY 29TH 7:52PM RUN #42 : bounty.append('[')
- v29 +-2 MAY 29TH 7:50PM RUN #41 ('): bounty.append('(')
- v24 +-d--33 MAY 29TH 7:44PM RUN #40 ('): bounty.append('{')
- v23 --10 MAY 29TH 7:42PM RUN #39 ('): bounty.append('{')
- v22 +-d--14 MAY 29TH 7:39PM RUN #38 ('): bounty.append('{')
- v21 +-22 MAY 29TH 7:35PM RUN #37 :ty(bounty, tk)
- v20 cell split MAY 28TH 1:22PM :pop()
- v10 +-24--8 MAY 22ND 2:34PM RUN #28 :rror('Unmatched target '+str(target)+"; "+str(bounty))
- v18 +-d4 MAY 22ND 2:30PM RUN #27

The rest of the code in the editor includes functions like `formatTokenList`, `getNewBounty`, `fulfillBounty`, and `processTokenList`.

Figure 11.5 Here we explore on-hover visualizations for the ambient version indicator that sits in the margin of each cell (Chapter 10 Verdant-1). Ultimately we did not implement the ambient version indicator and these interactions for Verdant-2 due to engineering time and resource constraints. We felt that the visualizations in the activity pane (sketches above) were more important to prioritize for our research questions about foraging for versions.

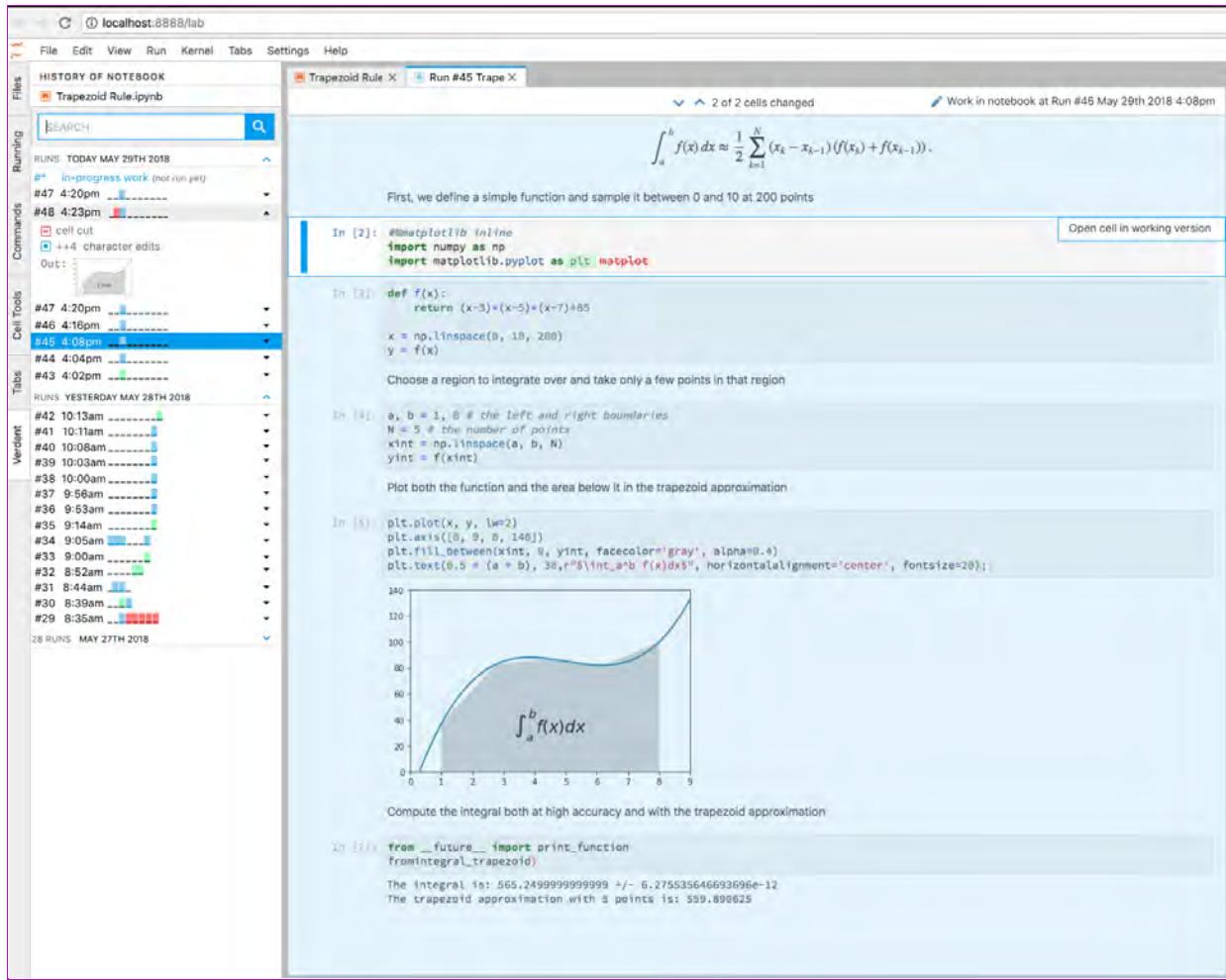


Figure 11.6 A concept sketch for the Ghost Notebook. When the user clicks on an older version of their notebook in the activity pane, a full read-only view of that notebook version appears in a new tab, with diff notation to highlight what content changed in that version. For this “Ghost Notebook” view we explored visual ways (such as the blue background) to help users clearly identify that they are looking at the past rather than their active notebook.

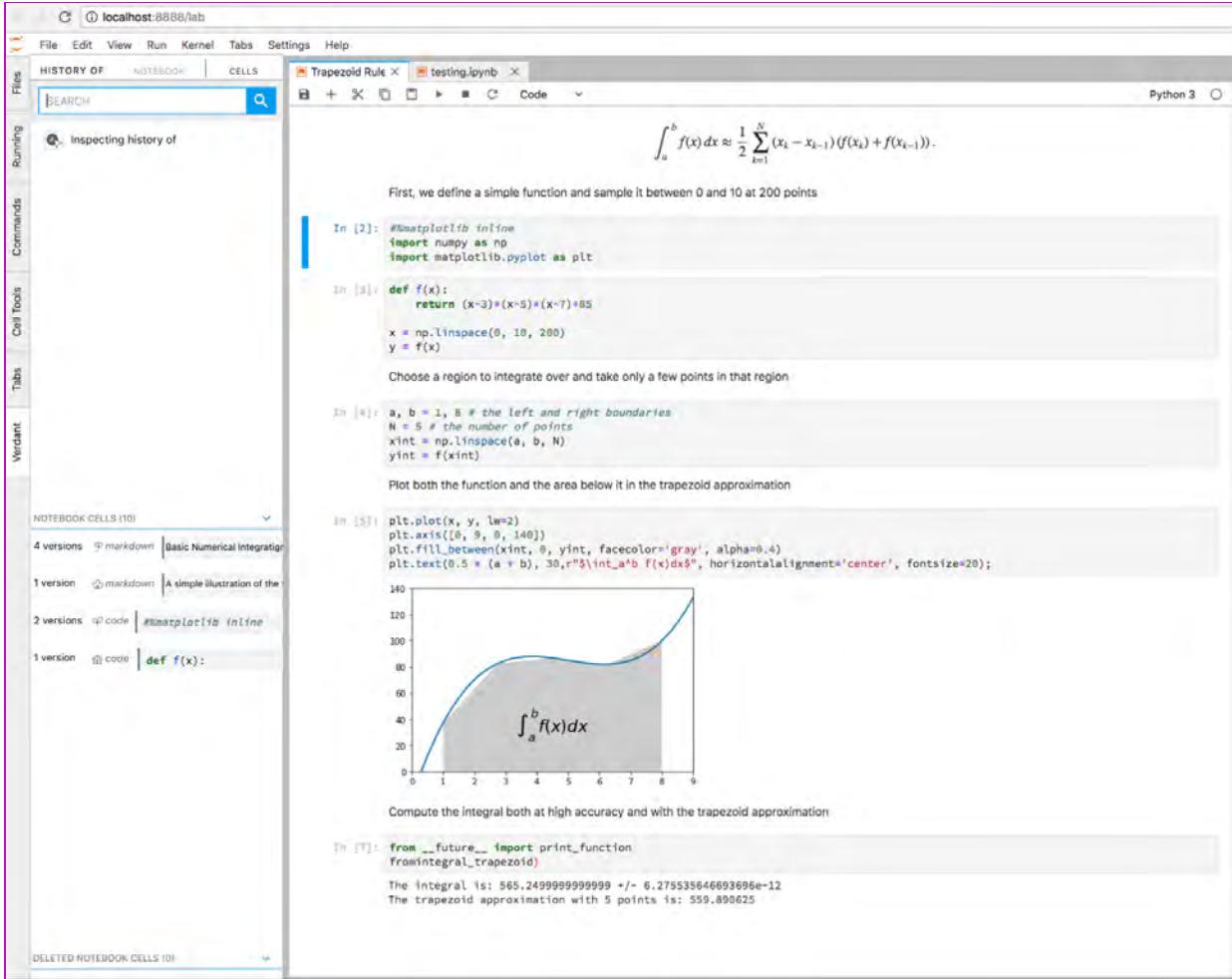


Figure 11.7 Early ideas for different features to help users search for artifacts: a search bar in the upper left, the version inspector (discussed below), and a table view summarizing cells and their versions (added later in Verdant-3).

SYSTEM: VERDANT-2

We began by recycling much of the same software design from Verdant-1, since the base need to record history is the same. Verdant-1 was built as an Electron App containing a Jupyter Notebook, which allowed a lot of freedom for hacking with designs but was at this point, not robust enough to translate into a real tool for use in the wild. We freshly re-implemented Verdant-1’s history collection and storage design in JupyterLab, and also improved on many efficiency details of history collection during the re-implementation. The goal of re-implementing in a broadly available platform like JupyterLab is that we need Verdant-2’s history functionality to be more robust to recording bigger projects and longer sessions that will provide us with that realistic scale of lots of history data. JupyterLab also provides us with side-pane screen space to test out UI designs situated in and side-by-side with the user’s notebook.

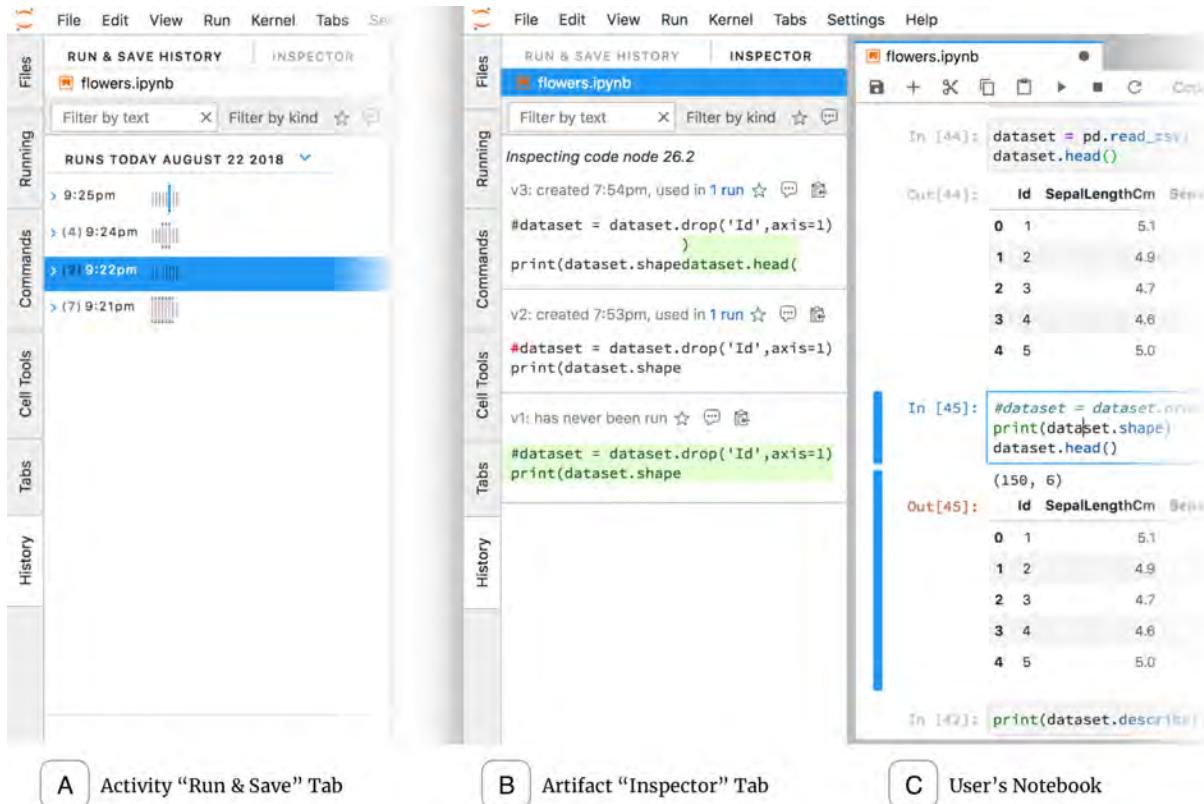


Figure 11.7 Overview of Verdant-2: the history interface is a side panel that sits to the left of the user’s notebook in JupyterLab. Verdant-2 has two view tabs. The activity view (A) shows ways to identify versions by date/time & location. The artifact “inspector” view (B) allows the user to click an artifact in their active notebook to see its full history.

Three key features in the Verdant-2 sidebar (Fig. 11.7 at A & B), support different foraging strategies users can employ to answer their questions. First, the Activity tab (Fig. 11.7 A) visualizes history shown by time and event so that the user can forage based on their memory about when and where a change occurred. A temporal representation of history is core to many other history tools like a conventional undo pane or a list of commits in Git. Second, the Artifacts Inspector tab (Figure 11.7 B) organizes history per artifact (cells & output artifacts) so that a user can forage based on what artifact changed and how a certain artifact evolved over time. Third, the search bar, shown in both tabs, offers a structured search through text queries and filters, which is useful when the users have a search keyword in mind or when their memories of when or where to start looking for an answer to their question are less precise. Each interface is next described in detail.

When? Where? Foraging in the Activity tab

Consider a use case where a data scientist has been iterating for a few hours on code for a regression, and asks “what were the beta values from the regression I ran earlier today?”

(Query Design Study, Chapter 4). Each artifact version in Verdant-2 is tied to a global-level event that triggered it, e.g., a run or save of the notebook. These are displayed in the Activity tab as a chronologically ordered stream of events (Fig. 11.8) so that the user can visually scan down to the rough date and time that constitutes “earlier today”.

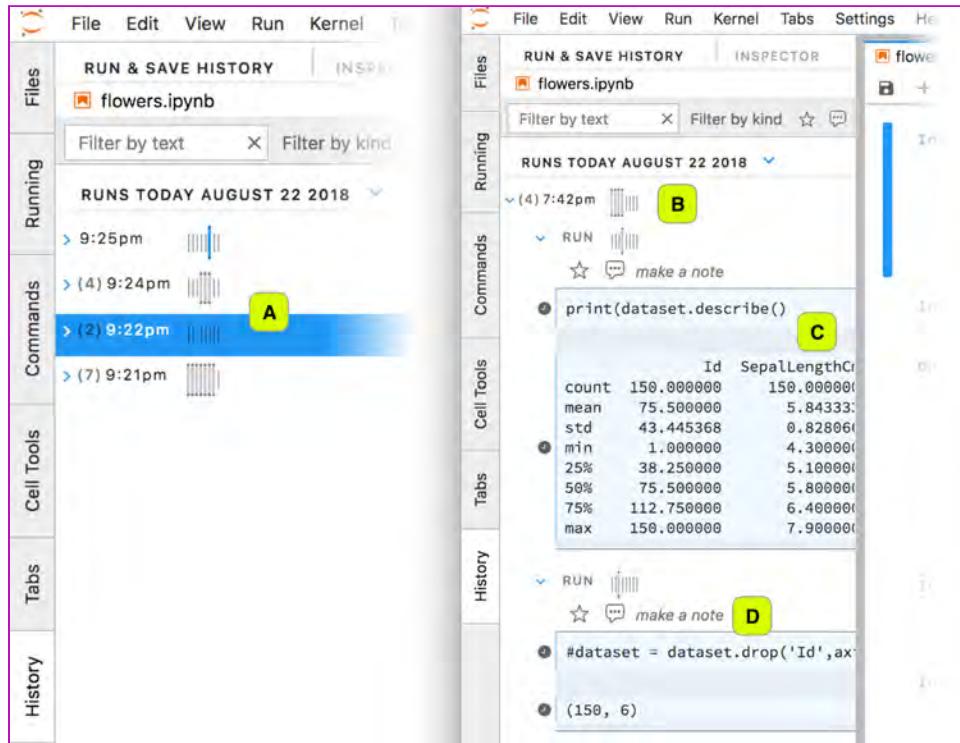


Figure 11.8. Run activity visualization of a notebook. In (A) each version of the notebook is visualized in a row with a timestamp and a minimap representing what changed. The user can expand each event by clicking on the caret beside the event’s row. In the expanded summary of an event (B) we show exactly which artifacts were changed (C) and allow the user to add custom annotations or bookmark that artifact’s version (D).

To give the visualization a bit denser information yield for foraging, run events that occur in non-overlapping cells within the same 60 seconds are recorded onto the same notebook version. This slightly reduces the granularity of notebook versions, but allows the user to see activity at a glance by minute, rather than by seconds.

Minute by minute may serve to spot recent activity, but a data scientist looking for “earlier today” will likely not recall the exact minute something occurred. However, a user might know where in the notebook they were working. Perhaps the answer lies during a time when many cells were added to the end of the notebook, or during a time when several cells in the middle were being edited and consolidated. We explored many designs to succinctly visualize where in the notebook activity occurred (Fig. 11.4), so that a user may rely on spatial familiarity with their own work to visually rule out an entire series of activity where irrelevant areas of the notebook were edited. Although it might be tempting to display textual labels, cells in a Jupyter Notebook are (currently) anonymous. The bracketed numbers to the left of cells in Jupyter notebooks (Fig.

11.7 C) are not stable and change as cells are added, deleted, or moved over time. To overcome these problems with names and to provide a tighter visualization, we were inspired by both a kind of tiny inline plot popularized by Tufte, called a sparkline [Tufte 2006], and a variation on a common code editor navigation visualization called a minimap²³. A conventional code minimap shows a miniature shape of the code file with colorful syntax highlighting so that a user can click based on spatial memory of where something occurs in their file, rather than reading exact lines. Prior work has suggested that notebook navigation limits the typical maximum number of cells in people’s notebooks to roughly 60 [Notebook Usage Study, Chapter 4, Rule et al. 2018]. In Verdant-2’s final minimap design (Figure 11.8 A), the notebook is flipped counter-clockwise to show the series of cells horizontally to conserve space. Each series of vertical lines after the notebook version number represents the entire notebook at that point in time. Each vertical line represents a cell and a taller bold line indicates activity: blue for cell edits, green for cell creation, red for cell deletion, and grey for running a cell without editing it. This representation makes it easy to spot such common cues as where cells have been added, or which portion of the notebook has undergone substantial editing.

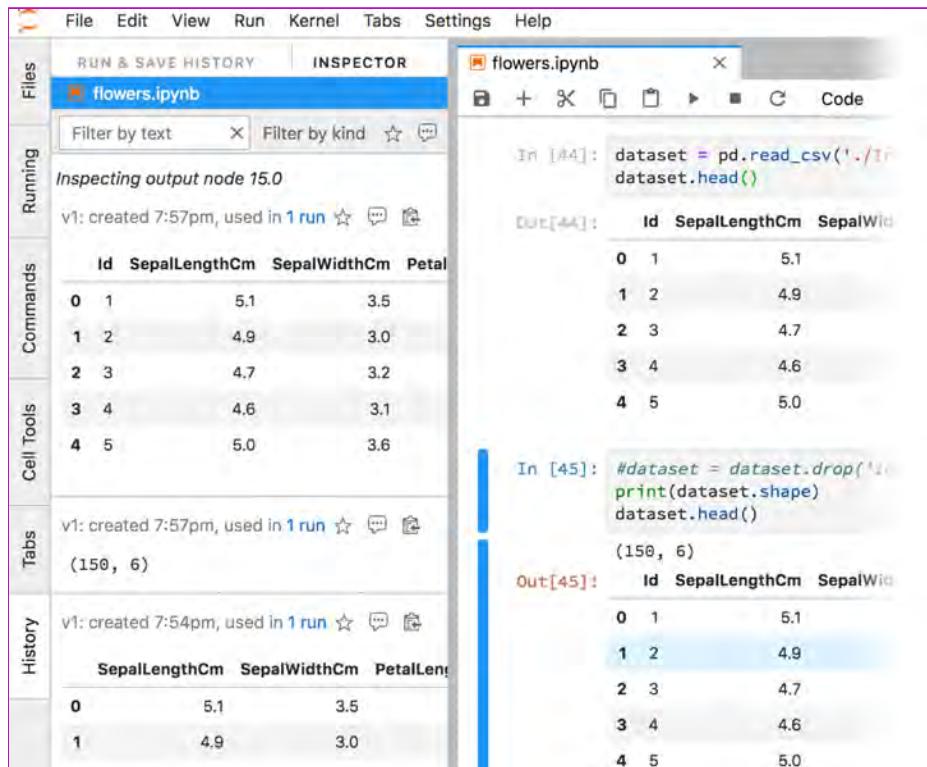


Figure 11.9 Inspector tab in Verdant-2 showing all the versions of a table output

What? How? Foraging in the Inspector Tab

Consider the case where the artifact is still in the current document, but has been changed since the older version the data scientist is looking for. Like preceding systems Variolite (Chapter 7),

²³ The exact history of code minimaps is unclear, but the underlying visualization technique appeared in the seminal 1992 software visualization paper Seesoft [Eick et al. 1992]. The code minimap visualization is widely supported in modern code editors, e.g., Atom and VS Code

Juxtapose [Hartmann et al. 2008] and Verdant-1 (Chapter 10), we assume that allowing a user to directly interact with the artifact in question is the fastest way for them to start foraging for an alternative version of that artifact. We adapt a design pattern from another context where there is a rich relational document full of sub-components each with its own sets of properties: a web page. With a browser’s style *inspector*, a developer can simply point to any element or sub-element on the active web page, and a browser pane then displays its style properties. This inspector interaction is tried and tested across all modern web browsers. We mimic this by, when the user double-clicks an artifact in the notebook, Verdant-2 provides a list of unique versions of that artifact (Fig. 11.9). The Inspector is its own tab in Verdant-2 (Figure 11.9), which is changed in subsequent versions.

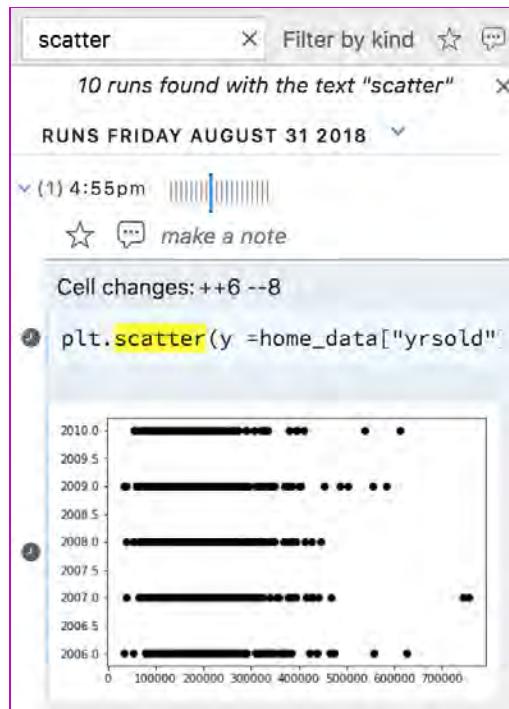


Figure 11.10. Search result for a history search of “scatter”

Searching with cues using the Search bar

Imagine that a data scientist is looking for all scatter plots generated within the last week. If that output is no longer in the notebook, the user will not be able to point to it in the Inspector (Figure 11.9). The Search bar is meant to give users a start when foraging for elements no longer contained in the notebook by supporting searching backwards through the history [Yoon et al. 2013]. By searching for “scatter” (Figure 11.10), the user receives a list of the matching versions of artifacts with that word in code, markdown, or textual outputs. The search mechanism includes various filters to help the user narrow-down results by artifact type or bookmarked artifacts.

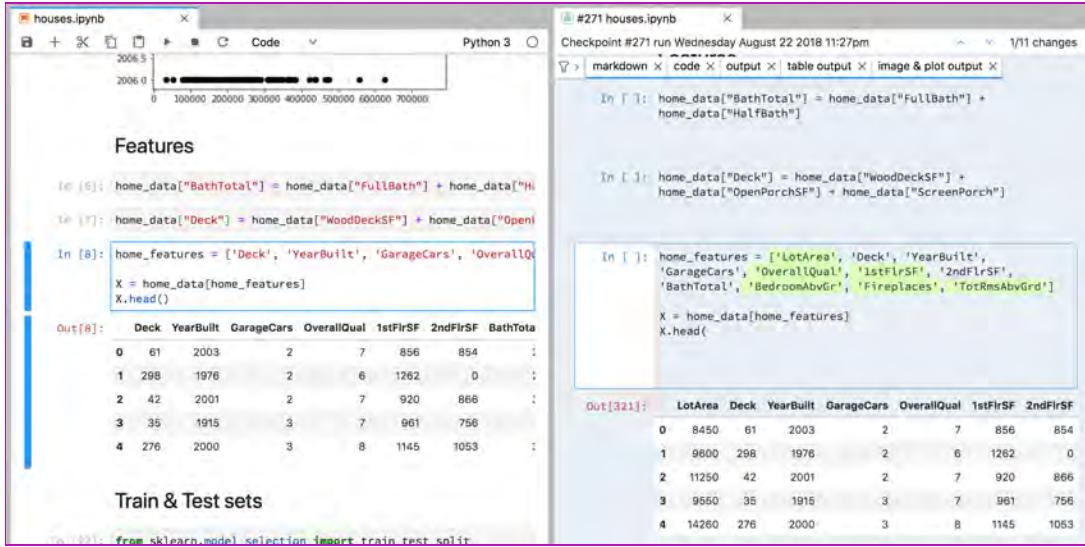


Figure 11.11. A past “ghost notebook” in side-by-side comparison with the user’s current working notebook

Resurrecting full revisions for context

Although our design criteria for the history tabs in Verdant-2 was to boil down information into small pieces for quick reference, more extended context is needed to answer some questions. If a data scientist wants to ask “what data was used to generate this plot?”, the code importing the data and how it was transformed to generate that plot may be spread across multiple locations in the notebook. Although using the Inspector tab, the user can view the detailed history of any artifact of cell/output size or smaller, we provide a different UI for notebook artifact versions, called a *ghost notebook*. This view allows the user to see a prior full notebook, and also shows the context of how specific smaller artifact versions are related to each other in that notebook. As shown in Fig.11.11, the ghost notebook is immutable, highlights where changes were made in that notebook version, and has a different background color from the user’s active Jupyter notebook to avoid accidentally confusing the two. The two notebooks can be viewed side-by-side, allowing the user to compare the older ghost notebook to their current notebook. The user can also open multiple ghost notebooks to compare across multiple historical states. An example use case for this would be to compare versions of a code file side by side [Codoban et al. 2015] to figure out “what changed?” between an earlier working version of the notebook and one that contains a bug.

JUPYTERCON SCAVENGER HUNT STUDY

The primary goal of our evaluation was to gather data about how the features of Verdant-2 assist or hinder data scientists in performing realistic foraging tasks. We had received positive feedback about our ideas from data scientists throughout the design process, so in this evaluation we sought task-based behavioral data to confirm or refute those positive opinions, and provide guidance for redesign. As Verdant-2 is an extension to JupyterLab, we coordinated closely with the Jupyter Project and ran our study at their JupyterCon2018 conference. JupyterCon annually gathers a concentrated group of data scientists, from a variety of sectors, with experience in computational notebooks, providing an opportunity to collect data from

professionals with a range of experience in a short period of time. We conducted the Notebook Usage Study (Chapter 4) in the same setting the previous year.

Challenges and Limitations of the Study

A conference setting presents considerable challenges to testing a complex tool intended for long-term use by expert users on their own code. A major difference between the primary use of a history tool, i.e., querying previous versions of your own code, and what we can study at a conference, is that we had to ask participants to find things in another person's code. Examining other people's code does happen in the real world, e.g., a manager of data scientists told us that he would find Verdant-2 useful for understanding his employees' thought processes, and professors sometimes grade student code on the basis of the process they employed as well as the end-product. Another difference is the skill with the tool itself that a data scientist would build up through long-term use. Both of these problems could be overcome through a longitudinal study. We conduct a longer-term study later in Part III of this dissertation.

Nonetheless, we believe the lack of skill with the tool, no knowledge of the code, and limited time to do the tasks can bring into stark relief any shortcomings in Verdant-2's UI design. The problems and virtues of Verdant-2's UI uncovered through performing tasks here give us a glimpse of how useful Verdant-2 would be at least for novice users and what we would need to do to improve it.

Study Materials

In order to create a realistic data science notebook history that both contained substantial experimentation and was simple enough for most participants to understand in a few minutes, we looked at some of the many data science tutorial notebooks available on the web. I created a notebook from scratch by following Kaggle's²⁴ machine learning tutorial level 1 on a housing selling-price dataset, followed by copying in and trying out code from Kaggle community notebooks from a competition with the same dataset²⁵. Creating a notebook this way, relying heavily on a variety of other programmers' code, was intended to reduce any bias that the study notebook history would be too specific to one programmer's particular coding style. The resulting 20 cell notebook contained over 300 versions.

The Tour.

We wrote eight instruction pages overviewing Verdant-2's features and how they worked. Each page contained a screenshot and annotation that drew attention to a feature and explained how it worked. It took less than 3 minutes to read through this document, shown as a web page, and participants could refer back to it at any time since both the tutorial and Verdant-2 were tabs in the same web browser.

²⁴ Kaggle is a popular platform where learners can practice data science skills and compete in data science competitions hosted by various companies and organizations.

²⁵ Tutorial: <https://www.kaggle.com/learn/machine-learning>, and competition: <https://www.kaggle.com/c/house-prices-advanced-regression-techniques>.

The Tasks

In our Query Design Exercise (Chapter 4), we asked data scientists “Given your own work practices, type as many [questions] as you can think of that could be helpful to you to retrieve a past experiment”. We converted some representative questions from the data scientists into tasks for the current study. Since the participants did not write the notebook, we had to substitute explicit goals for the memories a notebook author would have when setting foraging goals. For instance: “how did I generate plot 5” became a task “Find a notebook version that generated a plot that looks exactly like this [image shown]”, and “What data sources have I been using over the last month?” became a task “How many different data files has this notebook been run on?”. We generated 15 tasks across 4 task categories (Table 11.1).

Participants

JupyterCon2018 provided a User Experience (UX) “Test Fest” room where four organizations set up testing sessions and advertised its availability in the conference program, as slides in meeting rooms between sessions, by some presenters in their talks, and on social media. We recruited 16 participants who came to the UX room (referred to as J01 to J16). Due to equipment issues that arose during J11’s session, J11’s data will not be considered for analysis, leaving 15 participants. As shown in Table 11.3, participants’ jobs involved data science work across a wide range of domains.

Table 11.3. Work domain of data scientist participants.

Computational Domain	Participant
GIS or Earth science	J01, J08
Economics or Finance	J02, J09, J12, J16
Healthcare	J03, J04
Biology, Chemistry, or Physics	J15
HCI or Computer Science	J05, J14
Social Science	J10
Not reported	J06, J07, J13

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

Table 11.1 Tasks and number of each task category used

Category	#	Tasks
Notebook event	3	<p>A. Find the first version of the notebook</p> <p>B. How many cells have been deleted from the notebook during its history?</p> <p>C. How many runs did the author leave a comment on?</p>
Visual finding	3	<p>F. Find a notebook version that generated a plot that looks exactly like this [image1]</p> <p>K. Find a notebook version that generated a plot that looks exactly like this: [image2]</p> <p>L. Find a notebook version that generated a plot that looks exactly like this: [image3]</p>
Code finding	3	<p>H. At what time did the author last use a DecisionTreeRegressor?</p> <p>J. Find the code the author used to check for duplicate houses.</p> <p>N. How many different data files e.g., “data.csv” has this notebook been run on?</p>
Relation between artifacts	6	<p>D. Find the code in which the author explored compare mean absolute error with differing values of max_leaf_nodes</p> <p>E. In RandomForestRegressor(random_state=2), how many different values of random_state has the author tried?</p> <p>G. What was the lowest mean absolute error achieved when the author used a RandomForestRegressor?</p> <p>I. What was home_features equal to when the mean_absolute_error(val_y, val_predictions) was equal to or below 20,000?</p> <p>M. When “1stFlrSF” was not included in home_features, did that lower the value of mean_absolute_error(val_y, val_predictions)?</p> <p>O. When the author used DecisionTreeRegressor instead of a RandomForestRegressor, what parameters did they try for the decision tree?</p>

Procedure

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

QUANTITATIVE ANALYSIS

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

We used success rate as an indication of how well Verdant-2 supported the users in accomplishing tasks. The average success rate of the participants was 76% (median = 80%), which puts the evaluated version of Verdant-2 close to the average task success rate of usability tests across many domains [Sauro 2011]. Table 11.4 shows that more than half the participants succeeded at greater than 80% of the tasks they attempted and 20% succeeded at all of their tasks. Despite being asked to answer questions about a substantial notebook they did not write, having to forage through over 300 versions of that notebook, and having no experience with this complex tool, the majority of participants succeeded on the majority of tasks they attempted. For comparison, data scientists interviewed in Chapter 3 reported making many local copies of their notebook files. Imagine giving our participants over 300 files and asking them to answer a series of detailed questions about them. Many participants would have run out of time or given up. Even if our participants used Git, as discussed above, they would have had to learn complex command-line search tools and tasks involving graphic output may have been simply impossible. Thus we consider a median 80% success rate to be evidence that the design of Verdant-2 has promise but could be improved.

At this stage of development the overall success rate is interesting, but the differential success rate between tasks is more important for further design as it helps us focus on which tasks are more problematic for users. Turning to task success by task category (Table 11.5), the most difficult kind of task, “relationship between two artifacts”, which required hunting down and then relating versions of two or more separate artifacts, had the lowest success rate at 66%. Otherwise, there was no clear relationship between specific tasks we had *a priori* considered to be more “easy” or “complex” based on the number of steps required to accomplish the task. For

instance, the tasks at which participants had 100% success were task N: “How many different data files e.g., ‘data.csv’ has this notebook been run on?” (easy, at 3 steps) and task I: “What was home features equal to when the mean absolute error was below 20,000?” (complex, at 12 steps).

Table 11.4. Participant success by task category

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

QUALITATIVE ANALYSIS

We turn now to a qualitative usability analysis that investigates which features of the evaluated Verdant-2 UI were helpful and which may have hindered participants in accomplishing their tasks.

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

The data provided information at many levels, from comments on the tour, to buggy behavior, to complaints about low-level UI features like labels or icons that users found inscrutable, to issues with the high-level functionality. As an example of the latter, a data scientist in the Healthcare industry (J04), was concerned that Verdant-2 saved outputs, saying “We avoid ever checking data into a version control thing. If it was always saving the output, we wouldn’t be able to use it.” For the purpose of this chapter, we focus on just three problems: confusion about how to navigate within Verdant-2, the need for excessive scrolling, and participants resorting to brute-force looking through ghost books.

For the tasks with the lowest success rate, O and G in Table 11.1, participants would often click something and not know how to get back to where they had been. One-third of our participants articulated the equivalent of “How do I get back?” in these two tasks alone (J01, J05, J09, J12, J16). Looking more broadly, more than half of the participants (8/15) articulated this problem across 9 of the 15 tasks, with many more clicking around trying to get back without explicitly voicing their need.

To illustrate the scrolling problem, in Task F, the participants had to find a particular heatmap. The heatmap had been added sometime during the 300 versions, had been changed several times (the desired answer being one of those versions), then deleted. Of the 6 participants attempting this task, 5 immediately selected the correct feature (Activity tab) and the correct action (text search). J09 succeeded in 6 seconds because he had performed a graphic search task

before and knew to keep scrolling through the results. Four others succeeded within 3 minutes, but performed other actions in addition to the most efficient method (all tried ghost books; 2 tried the Inspector Tab) and those actions provide clues to better design. Consistent with Information Foraging Theory [Pirolli 2007], these detours suggest that having to scroll too long before finding promising results causes people to lose confidence in the information patch and abandon it.

At a higher level, we observed many participants resorting to a brute-force search. “It’s obvious if I looked at all of these [ghost books], then I’d know the answer, but there’s got to be a smarter way to do this.” (J06) They opened up one ghost book at a time until they reached the solution or became so frustrated they switched to a different foraging tactic (such as searching with a different term) [Piorkowski 2015] or else quit the task altogether: “I found 22 things... I can find it, but I’m not sure I have the patience.” (J03). One participant (J10) to our surprise, sat for a full 6 minutes and read through 39 different ghost books before reaching an answer. Although none of the tasks actually required using brute-force search of ghost books, it is a problem that users got to a point where they thought brute-force was the only solution available to them.

Our evaluation collected task-based behavioral data as well as opinions and suggestions from professional data scientists. This left us with a trove of bugs to fix, UI elements to tweak, and more areas to redesign than we could present here. Our next major priority in design was to try and fix the navigation issues participants had encountered, while preserving the effective parts of Verdant-2’s design.

CHAPTER CONCLUSIONS

In Verdant-2 we focused on designing features that enable practitioners to answer specific questions from history at scale. The kinds of questions Verdant-2 supports are those we collected from practitioners in the Query Design Exercise (Chapter 4), which we then categorized into specific kinds of memory cues like visual appearance, datetime, location, or keywords. Each cue has an accompanying feature in Verdant-2: minimaps for location, the activity view for datetime, the artifact view for appearance, and the history search for keywords.

Ultimately the goal of Verdant-2 is: given a pile of “stu and stu and stu” (Chapter 4) users can quickly grab useful experiment information out of that pile. Our Jupytercon Scavenger Hunt Study showed that Verdant-2 comes reasonably close as an initial prototype. Many participants *were successful* with Verdant-2 in retrieving substantive experiment information out of a long history. However, there were design aspects, other than the memory cues we had carefully designed for, that our study revealed that could make a history search much easier or more difficult for the user. Here I will list some of the key design takeaways in what worked and what didn’t in our design for Verdant-2:

- + **Structure history to minimize scrolling through lists:** Consistent with prior work on information foraging in programming, we saw in our usability study that users became frustrated if they believed that they needed to examine every version in a list to identify the right one. In prototyping (and redesign after this study) we have found that strategies like binning, previewing just a snippet of a version, or being more careful

about how history is collected in the first place can all reduce the amount of information a user needs to visually browse through for a given view of history.

- + **Give the user multiple ways to find the same thing based on what they remember:** since Verdant-2 has multiple ways of searching and visualizing history, there is no single “right” way to find something. We saw this benefit users in our study because regardless of which avenues a participant tried, they were able to reach the same information. We also observed that different users had different preferences for relying on different search features.
- + **Treat visual finding for visual artifact history differently:** A missing memory cue that we did not explicitly design for in Verdant-1 is visual memory. We assumed that textual artifacts like code and visual artifacts like plots could be found roughly the same. However in practice we found that plots often do not have simple descriptive keywords in the code that accompanies them, so they cannot be found with a textual keyword search. In our usability study, visual finding was one of the least successful kinds of tasks in Verdant-2. It is worth designing search features specifically for finding plots, images, and other visual-only artifacts.

Chapter 12: Design and Engineering Iterations for Robustness

Featuring Verdant-3 & Verdant-4

Research done in collaboration with Brad A. Myers, Bonnie E. John, and Abhishek Vijayakumar

INTRODUCTION

Following our Jupytercon Scavenger Hunt Study, our next major research goal was to iterate and then test out our history interactions with data scientists doing real exploratory programming of their own. Putting Verdant to *real usage* would allow us to test major hypotheses in this dissertation: *Can automatic history tooling benefit a person’s experimentation? Can we design history such that a person can quickly answer questions about their past experimentation?* To reach this goal required substantial investment into engineering hours and design to make Verdant more usable and robust. Real usage for sustained programming activity requires our history system to be less of a research prototype and much more a real product. This chapter details the iteration we took on Verdant to prepare it for real deployment in our next study.

VERDANT-3

First we list major usability issues from Chapter 11’s Jupytercon Scavenger Hunt Study that our Verdant-3 redesign attempts to address:

1. Confusion about how to navigate between Verdant-2’s different views & features.
2. The need for excessive scrolling through information.
3. Participants resorting to brute-force looking through many ghost books.
4. No *names* for artifacts or versions that users can write down or reference.
5. Unclear that the tabs in Verdant-2 are actually tabs.

To address Issue #1, much of this redesign attempts to “disentangle” all the views of Verdant-2 such that each major feature of Verdant has a clearly separate view, with clearer and easier navigation connecting features. First, we change the top-level organization of the interface:

Verdant-2	Run & Save History tab	Inspector tab	
Verdant-3	Activity tab	Artifact tab	Search tab

The “Run & Save History” tab is renamed to “Activity” simply because Verdant collects history at many different notebook events (see Chapter 9) and not just run and save. The “Inspector”

tab is renamed to “Artifact” so that the title is more about the *content* a user will see rather than the specific “inspector” interaction used on that tab. Previously, the search bar “filtered” views in the Activity and Artifacts tabs (Figure 11.10), but participants often got confused over whether the search was on or off (a mode error). To remove that confusion we separated search into its own tab with its own view.

Other redesign details to address the other usability issues are detailed in the figures of Verdant-3’s design below.

Screens

The screenshot shows a search interface with a search bar at the top containing the query "garage". Below the search bar is a list of search results categorized by artifact type. The results are grouped into three main sections: "code artifacts", "markdown", and "output". Each section has a header with a dropdown arrow and a count of matches. The "output" section is expanded, showing a sub-section for "VERSIONS OF OUTPUT 5" with a link to "V10 OUTPUT 5, NOTEBOOK #36". At the bottom, there is a table with columns: TotRmsAbvGrd, OverallQual, and garageCars. The data rows are: (8, 7, 2), (6, 6, 2), (6, 7, 2), and (7, 7, 3).

TotRmsAbvGrd	OverallQual	garageCars
8	7	2
6	6	2
6	7	2
7	7	3

Figure 12.1 In (A) and (B) the user can search history using keywords and filters as in Verdant-2. To reduce excessive scrolling, we clustered the search into accordion bins by artifact type (C). In (D) we include links with clear names so that the user can open up the Artifact Detail View or Ghost Notebook from a search result.

Activity	Artifacts	
TODAY JANUARY 2 2019	A	
10:07pm Run	#32	
10:06pm Cell removed	B #31	
10:05pm Cell added	#30	
10:04pm Run	#29	
10:03pm Run 10:04pm Save	#28	
10:02pm Cell added	#27	
10:02pm Runs	D #26	

Figure 12.2 The Activity view is simplified from the one in Verdant-2 (Figure 11.8) to prioritize clarity. To reduce the need to scroll, we changed our history model for event clustering such that more events refer to the same notebook (e.g. #28 above has 2 events run and save). The date title takes less space (A), and we add the particular event type to time descriptions in (B). Also in (B) we show a version number for each notebook version for the user to reference and more easily remember which version they are looking at. In (D) we also simplify each event row so that it no longer previews changes in a dropdown view. Instead, if users click the event row a Ghost Notebook opens up for that notebook version, so that the user can see those same changes in context.



The screenshot shows a user interface for managing artifact versions. At the top, there are tabs for 'Activity' and 'Artifacts', with a search icon. Below the tabs, the path 'NOTEBOOK > CELL 12 > ASSIGN 248' is displayed. A small circular icon with a play symbol is next to the path.

The main content area displays three code versions:

```
V3 ASSIGN 248 FROM CODE CELL 12, NOTEBOOK #34
ind="quicksort", ascending=False).drop_duplicates()

V2 ASSIGN 248 FROM CODE CELL 12, NOTEBOOK #32
= s.sort_values(kind="quicksort", ascending=False)

V1 ASSIGN 248 FROM CODE CELL 12, NOTEBOOK #29
so = s.sort_values(kind="quicksort")
```

Below the code, the section 'OUTPUT 12' is shown, containing the following table:

	Maintenance	1.000000
GarageCars	GarageArea	0.882475
GarageYrBlt	YearBuilt	0.825667
TotRmsAbvGrd	GrLivArea	0.825489
TotalBsmtSF	1stFlrSF	0.819530
SalePrice	OverallQual	0.790982
GrLivArea	SalePrice	0.708624
2ndFlrSF	GrLivArea	0.687501
YearBuilt	Maintenance	0.685644
BedroomAbvGr	TotRmsAbvGrd	0.676620
BsmtFinSF1	BsmtFullBath	0.649212
YearRemodAdd	GarageYrBlt	0.642277
SalePrice	GarageCars	0.640409
Maintenance	OverallCond	0.638171
GrLivArea	FullBath	0.630012

Figure 12.3 To reduce the need to scroll, the Artifact Detail view's text and content layout is changed from Verdant-2 (Figure 11.9) in order to fit more versions on screen. To help a user more clearly reference how hierarchical artifact versions relate, we add explicit names for the cell number and notebook version number (e.g. *from Code Cell 12, Notebook #29 above*). Finally, for code artifacts we include the output versions of that code below, to reduce the need for the user to navigate back and forth between output and code.

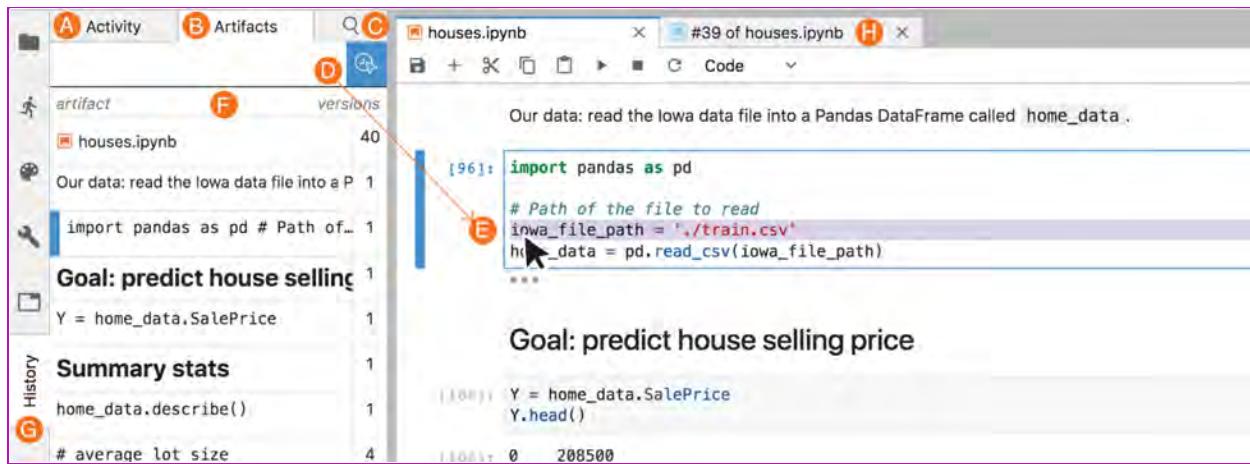


Figure 12.4 When the user has not inspected a particular artifact, Verdant-2 showed a blank screen in the Artifact tab. Now the Artifact tab displays an Artifact Table view (F) which summarizes all cells currently in the notebook and how many versions each has. If the user clicks on a row in the table, that opens the Artifact Detail View (Figure 12.3) which is the same end result as if the user had used the inspector interaction (by clicking D then a place in their notebook E) to get the Artifact Detail View of that artifact. This table is meant to reduce unnecessary scrolling through the notebook. To do that, each row is a preview of just the first line of a cell in the notebook, so that the user can get a sense of which cell is which in a shorter format than the entire document.

To address the issues of users scrolling too much or brute-force searching through Ghost Notebooks, we also made changes to the underlying history model. For instance, as discussed in Chapter 9, we applied some clustering heuristics to combine versions and reduce the total number of notebook versions Verdant produces in the first place, which in turn reduces how many versions a user needs to view.

Where did those features go?

Finally, it is important to note that we *removed* multiple smaller features of Verdant between Verdant-2 and Verdant-3. Search filters were removed, as was the ability to bookmark or take notes on a particular feature. However, this is not because we had any evidence or belief that these were bad features. Rather, again due to limited engineering resources we simply needed to reduce the number of UI features that would be deployed with Verdant. We removed features that we believed were secondary and not immediately critical for letting users effectively search history in Verdant.

DEPLOYMENT PILOT

We conducted a pilot study of Verdant-3 in a classroom setting, where students in a data-centric course were asked to install Verdant for use during a single homework assignment. Verdant-3 was part of the assignment for the whole class, but students were given a consent

form which allowed them to decide whether or not they consented to having their data used in our research. Students were given instructions for installing Verdant-3 and JupyterLab on their own computers, and were provided with support over Piazza or in person during office hours for any issues they encountered during installation. Students were given over a week to complete the assignment, and at the end of the assignment were given the option of donating their Verdant log data for our research study. There was no compensation or credit given for donating log data. All students were given the same opportunity of a small amount of extra credit for completion of a short feedback survey.

Although our logs indicate that students barely used Verdant-3's UI, perhaps due to the brief and constrained nature of the coding assignment, the history logs passively collected by Verdant-3 appeared to work fairly well. The ease of installation and relatively few problems reported were encouraging. This pilot did, however, uncover some data corruptions and bugs that occurred in unconstrained usage. We used data and history logs from this pilot to improve Verdant's robustness and design. Logs from these real users allowed us to further refine our designs by being able to simulate real usage based on these logs rather than using the "toy" histories we had created ourselves as a team for past design iterations. This led to some significant design changes for Verdant-4, the final version of Verdant we created in preparation for our next full study.

VERDANT-4

While the Jupytercon Scavenger Hunt Study discussed in Chapter 11 provided evidence that Verdant does have promising visualizations and features, users struggled to understand the relationships between different artifacts and versions. While we attempted to simplify and clarify the relationship between different views in Verdant-3, further heuristic evaluation and usability walkthroughs conducted within our research team demonstrated to us that Verdant was still too confusing and difficult to navigate. Thus the goals of Verdant iteration Verdant-4 were:

- Improve simplicity and clarity of all screens.
- Improve navigation and add more pathways among features.

We believed one more redesign was worth it before deploying Verdant, since the success of our history tooling hinges on data scientists being able to quickly and effectively navigate history with our tool. The following screens summarize changes made and the final version of Verdant before deployment.

Screens

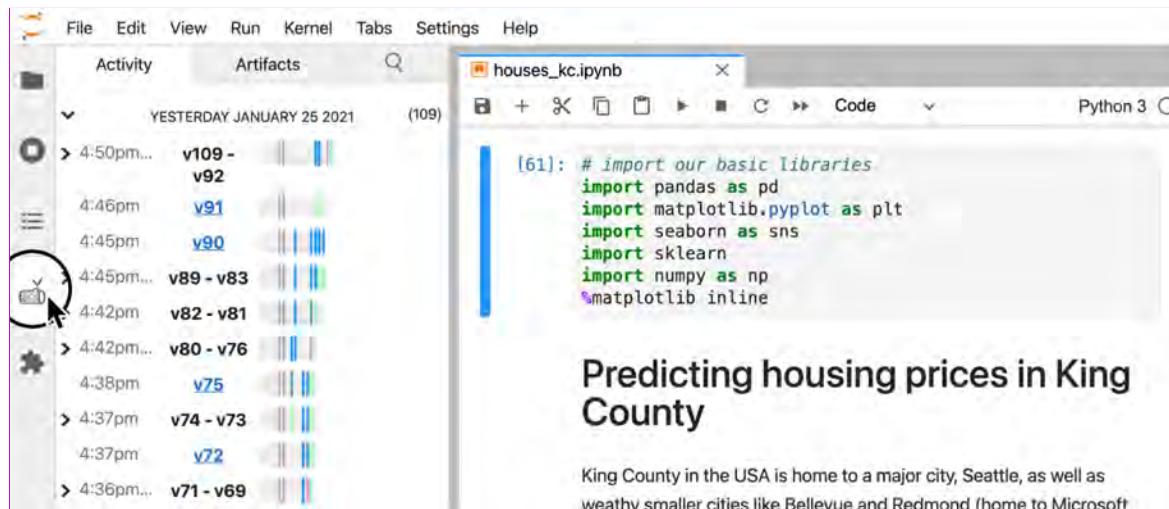


Figure 12.5 We changed the button which opens Verdant to be a log icon (circled with cursor) because JupyterLab updated its own style from text labels to icons in the sidebar. We also further simplified the Activity view and added more clustering of versions in order to fit more versions on screen to reduce scrolling. Event text labels like “run” or “save” were removed, since they took up space and were redundant information with the minimaps. We restyled the minimaps to make them easier to see the colors of the tics and compare across rows. Finally we restyled the notebook version labels to make it clearer to users that they are versions with a v instead of a # prefix..

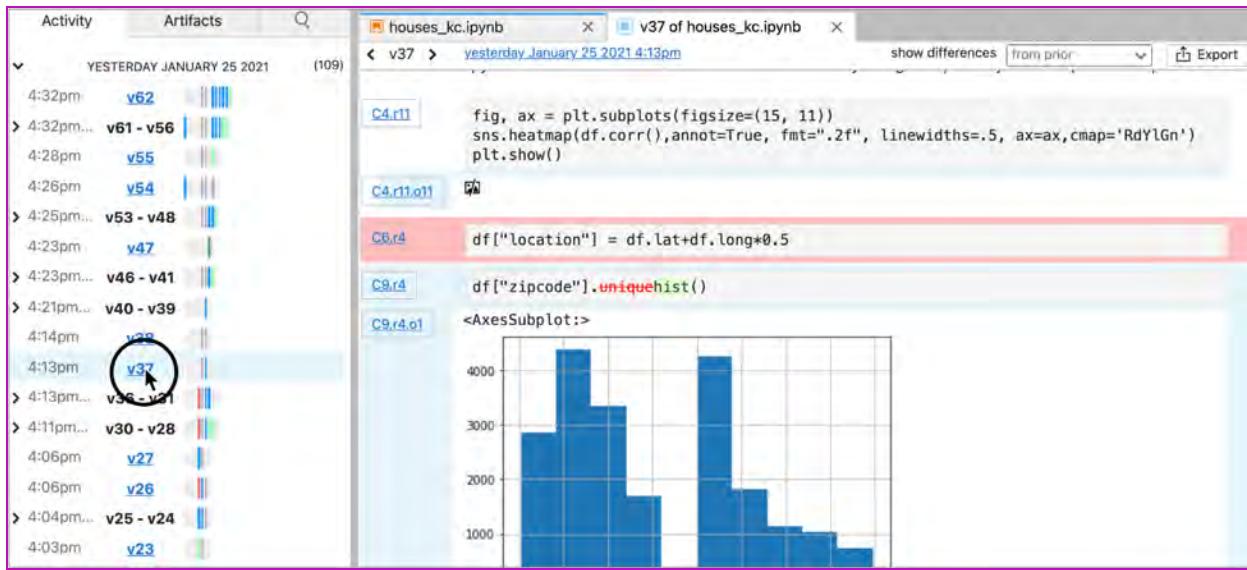
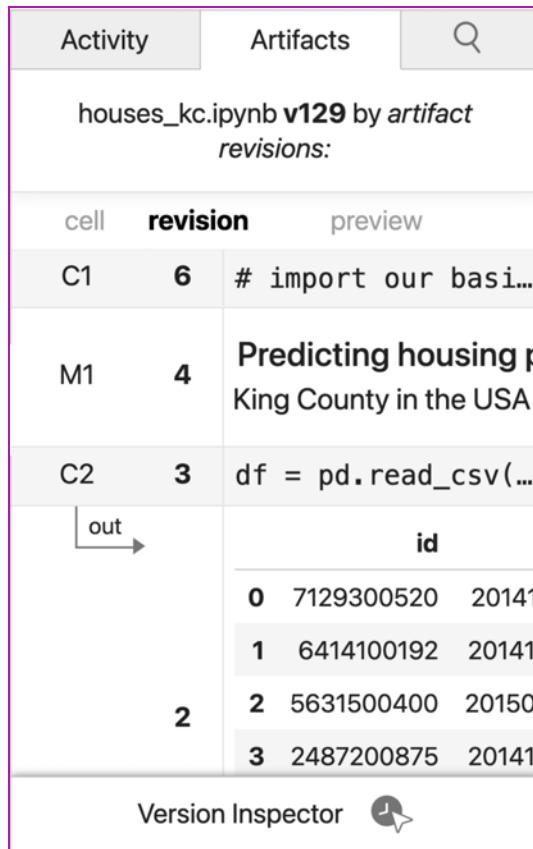


Figure 12.6 To make it more obvious how to open a Ghost Notebook, we changed notebook version labels to be blue hyperlinks that will open up that Ghost Notebook on the right. We also added the same style of blue hyperlink in the Ghost Notebook so that if a user clicks a cell (like code cell C6.r4 above) it opens the artifact view of that cell to show all of its versions.



The screenshot shows a user interface for managing artifacts from a Jupyter Notebook. At the top, there are tabs for "Activity" and "Artifacts", with "Artifacts" being the active tab. A search icon is also present. Below the tabs, the title of the artifact is displayed: "houses_kc.ipynb v129 by artifact revisions:". The main content is a table with three columns: "cell", "revision", and "preview". The table contains the following data:

cell	revision	preview															
C1	6	# import our basic...															
M1	4	Predicting housing price in King County in the USA															
C2	3	df = pd.read_csv(...)															
	out	<table border="1"> <thead> <tr> <th></th> <th colspan="2">id</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>7129300520</td> <td>20141...</td> </tr> <tr> <td>1</td> <td>6414100192</td> <td>20141...</td> </tr> <tr> <td>2</td> <td>5631500400</td> <td>20150...</td> </tr> <tr> <td>3</td> <td>2487200875</td> <td>20141...</td> </tr> </tbody> </table>		id		0	7129300520	20141...	1	6414100192	20141...	2	5631500400	20150...	3	2487200875	20141...
	id																
0	7129300520	20141...															
1	6414100192	20141...															
2	5631500400	20150...															
3	2487200875	20141...															

At the bottom of the table, there is a "Version Inspector" button with a magnifying glass icon.

Figure 12.6 We updated the Artifact Table View to add previews of outputs and be overall easier to read. We iterated with paper prototypes to test how well people could understand what this table is showing. To make things clearer we added headers to the table “cell”, “revision”, and “preview” and added the cell names “C1, M2, C2...”. The label “revision” was bolded because people frequently missed the meaning of the revision numbers during paper prototyping. Finally we made the title of the table a full phrase to better convey that this is showing the contents of a specific notebook version (v129 above).

The screenshot shows two views of a software interface for managing artifacts.

Left View (Artifact Table View):

- Header: Activity, Artifacts, Search icon.
- Title: houses_kc.ipynb v133 by artifact revisions:
- Table:
 | cell | revision | preview |
| --- | --- | --- |
| C1 | 6 | # import our basic... |
| M1 | 4 | Predicting housing price in King County in the USA |
| C2 | 3 | df = pd.read_csv(...) |

Right View (Artifact Detail View):

- Header: Activity, Artifacts, Search icon.
- Breadcrumb: NOTEBOOK > CODE CELL 13
- Row highlighted with a red circle and arrow: C13.r16 created in Notebook v127 1:25pm Jan 29, 2021
- Code preview:


```
x = np.array(df[['grade', 'sqft_living', 'price']])
y = np.array(df['price']) # what's the relationship?
```
- Other rows:
 - C13.r15 created in Notebook v125 1:25pm Jan 29, 2021
 - C13.r14 created in Notebook v124 1:25pm Jan 29, 2021

A large arrow points from the left view to the right view, with the text "Click Notebook in the nav bar to go back to the table view".

Figure 12.7 We worked on making navigation between the Artifact Table View (left) and the Artifact Detail View (right) more obvious. When the user clicks on a row in the Artifact Table View OR clicks on an artifact using the Inspector interaction in their current notebook OR clicks on an artifact in the Ghost Notebook, the view switches to showing all versions of that cell in the Artifact Detail View. To get back to the table view, we show a breadcrumb menu at the top of the Artifact Detail View.

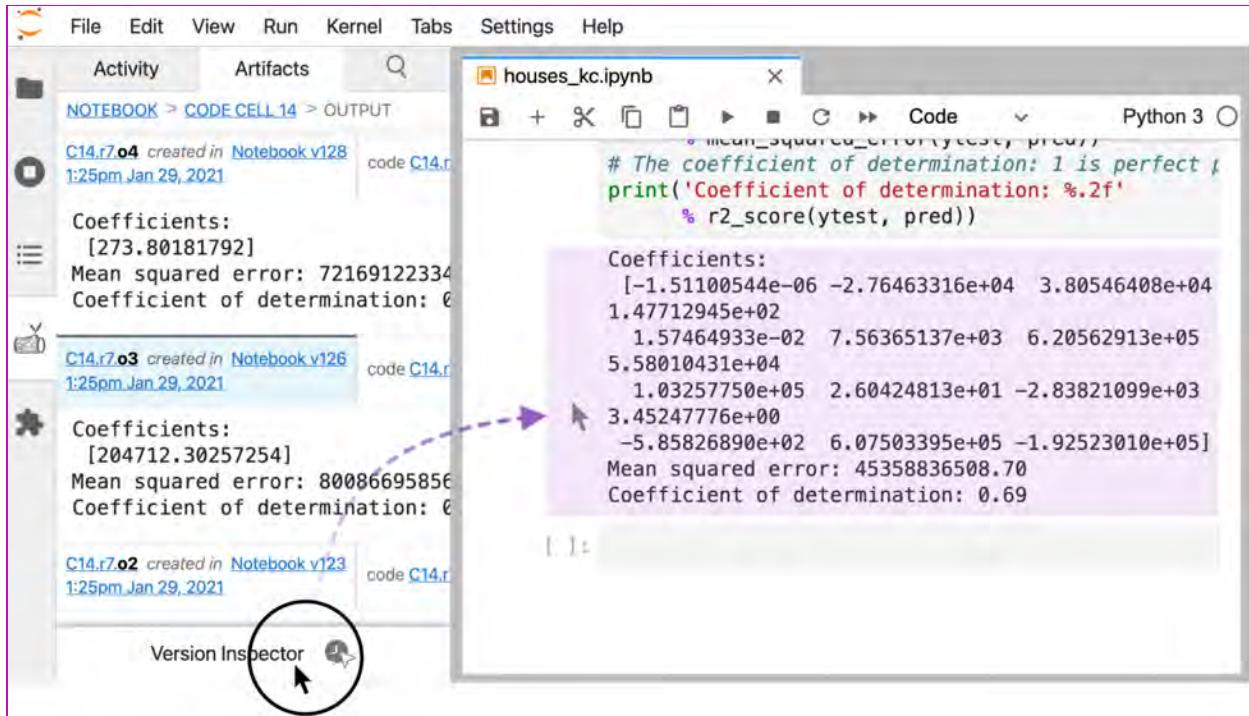


Figure 12.8 We made the Inspector interaction button larger in an attempt to make it more obvious to users. We also redesigned the Artifact Detail View to make it more obvious how to get back and forth between code and its output.

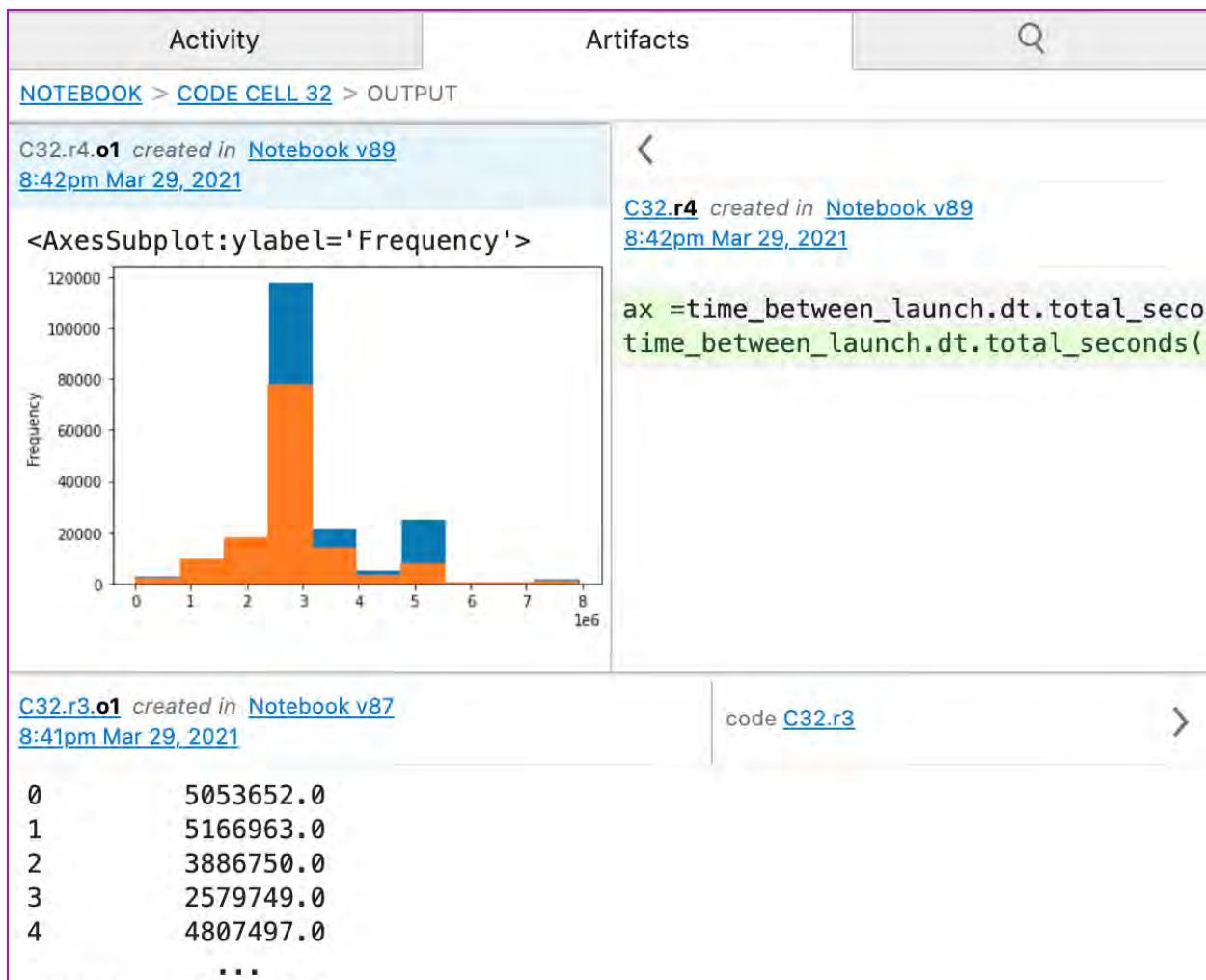


Figure 12.9 In Verdant-3 we had just put all versions of output underneath all versions of the code that generated it, however this did not tell users which code versions go with which output. To solve this problem we tried to add more informative labels so that a user can easily link to what code generated which output. The user can also expand the history pane horizontally to see code and output pairs side-by-side. Shown above, the first output version is shown with its code side-by-side. Side-by-side view can be opened or closed by clicking on the far right > button for each version.

New Features for Finding Visual Output

In the Jupytercon Scavenger Hunt Study, we saw that Verdant's search feature did not work particularly well for visual output such as plots, since it is a textual keyword search. To address this issue sufficiently for our next study, we introduced 2 new features which are something of a "hack". First, we added a number of special keywords for the search, such that if the user types in "plot", "image", "chart" or related words, we simply return all image outputs. This is not an ideal approach since if the user types in "bar plot" the search will return *all* plots, however we decided that a more advanced search was out-of-scope for our current research goals, and it was preferable to have many false positive search results rather than any false negatives.

Second, following our Deployment Pilot, we made a major performance improvement to have all visual output saved in an external history folder (see Chapter 9 for details). Verdant-4 saves all images in this folder with a particular naming convention for runtime retrieval. For instance in Figure 12.10 below, `output_10_33_0.png` refers to the first output of the 33nd version of code cell #10. As something of a navigation “hack” to browse through visual output, users can browse plots in the external folder on their computer, and then search the file name of that output in search to retrieve its history. This allows a data scientist to take an image, enter its file name, and then access the historical code that generated that image to reproduce it.

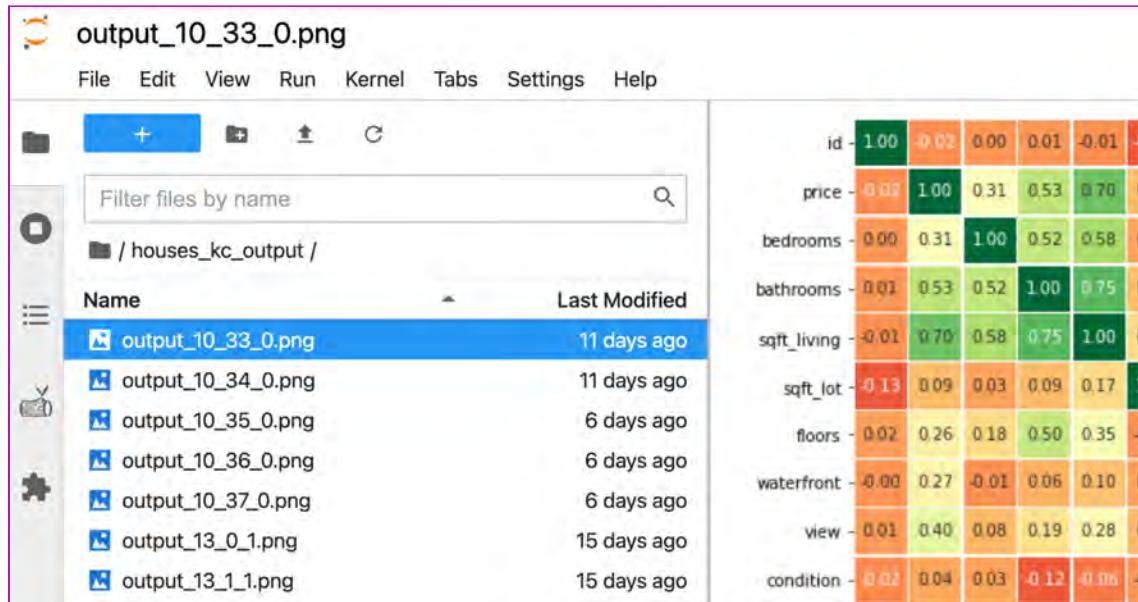


Figure 12.10 The external output history folder Verdant-4 creates, shown here in JupyterLab’s file browser.

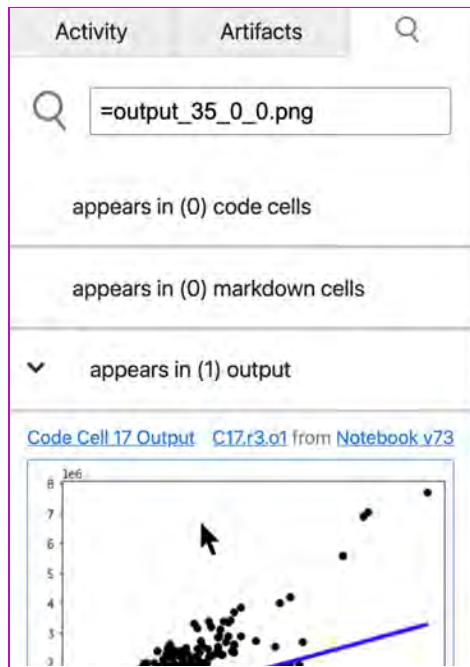


Figure 12.11 The user can search an image file name to retrieve its associated history and the code and notebook that generated it.

System Deployment

Verdant-4 was deployed in January of 2021 as an npm package such that any JupyterLab user can download and install Verdant directly from the extensions tab of JupyterLab. It has since been downloaded by 1,689 people in the wild as of August 17, 2021. Although we included a usability feedback survey in the link with Verdant, it has only gotten 2 responses from users, who both rated Verdant high for usefulness on a Likert scale, and moderately positive for the clarity and ease of use of Verdant.

<https://github.com/mkery/Verdant>

<https://www.npmjs.com/package/verdant-history>

CHAPTER CONCLUSIONS

At the conclusion of the systems design section of this dissertation, where have we ended up? We started in Chapter 7 creating inline history access for just a few code snippets with Variolite. By comparison, with Verdant-4 (some 5 years of research and iteration later) the user can access specific history of any snippet of any kind of content in their document, and recover information about their experimentation among hundreds of versions. Each iteration and new system in this part of the dissertation moved forward our understanding of the design space. Our design questions also evolved over time towards tackling more and more complex issues for history support. Even in this chapter, while Verdant-3 and Verdant-4 appear as a cumulation of many, many, small tweaks from the core design presented in Verdant-2, our design questions

for Verdant-3 and Verdant-4 had crucially shifted since our design work in Verdant-2. In Verdant-2 we were most concerned with *retrieval* of a specific version of a specific artifact: can a user recover a specific thing from a messy pile of history? In our Jupytercon Study at the end of Chapter 11, we observed that to actually answer history questions, users need to *understand* experimentation from history logs and how artifact versions relate over time. Where understanding the relationship between artifacts was the worst-performing kind of task in Verdant-2, *clarifying history relationships* became the focus of design efforts in Verdant-3 and Verdant-4. Clarifying the relationship between features of Verdant, and between artifacts in history also served to clean up some of the usability issues that had plagued users of Verdant-2. Next, in the final part of this thesis work, we test Verdant-4 out with practitioners in realistic usage to see how effective our designs are. Although we could have repeated the same protocol from the Jupytercon Scavenger Hunt Study to test Verdant-4, all of our work in engineering and the deployment pilot moved us to our next goal in research. In the Verdant Study, discussed next, we put history support into action in realistic exploratory programming data work.

Part III: Putting Experiment History into Practice

INTRODUCTION TO PART III

How will we know if we've met our design goals? How will we know if our prototypes are *actually effective* support for history of exploratory programming data work? First, we need to more closely define “effective”. This dissertation explores multiple areas within the design space of history support, but our furthest developed prototype, Verdant, is designed specially to allow users to quickly recall specific information from their work history. So, we consider Verdant to be effective *if* a data worker gets asked realistic questions about their work, *and* is able to use Verdant to provide history evidence to answer those questions. Additionally, since we know that recalling specific experiment facts out of real history logs can be slow and hard to do (Chapter 10), we will consider Verdant to be effective *if* a data worker can answer realistic analysis or modeling history questions *quickly* and with low effort.

In Chapter 14 we describe our study design to test Verdant in as realistic scenarios as possible. Although Verdant is a publicly deployed tool and a field study would be possible, we opt instead to do a more controlled “quasi-field study” in which we give data practitioners the same data work prompt, dataset, and tool environment, which we strive to make as realistic as possible. Controlling the environment and tasks gives us more consistency to compare how Verdant performs between practitioners with different domain expertise and experience levels. We design a two-part study in which practitioners first do real exploratory programming data work using Verdant, and then in a follow-up session, answer history questions using Verdant.

For our Results in Chapter 15, we find that all participants successfully found history evidence to answer every question but one posed to them, for a 98% success rate. Moreover, on average participants needed just 1 minute, 26 seconds to find the history information they were looking for. We discuss interviews with participants in which we dig into areas for improvement for Verdant. Finally, we discuss participants’ views on the value and viability of Verdant’s style of history in their own data work practice.

Chapter 13: Designing a Realistic Usage Study

INTRODUCTION

Ideally, to test Verdant, we would like to be a fly-on-the-wall observing real practitioners use Verdant during their real daily analysis work, and then later ask those practitioners to answer questions about their work using Verdant’s history. In real life, however, there are major barriers to access to real data work: NDAs, confidential data, restricted data access, specialized workflows and tooling, proprietary analyses, restrictions to recording audio or video in real workplaces, and so on — the list of research barriers quickly piles up to make a field study, if not impossible, then certainly questionably fruitful for this point in our research. On the other end of the spectrum, a traditional lab study controlled evaluation of Verdant would be too artificial to meet our research goals, since we seek to not only measure the efficacy of Verdant, but also its viability to help data workers in real world experimentation practice. To give ourselves enough high-quality research data to test Verdant, we blend lab study and field study approaches to create a “quasi-field study” where we provide practitioners with a fixed dataset and environment, all while endeavouring to create as much realism and external validity as possible in our study design.

TWO-SESSION DESIGN

We use a two-session study design to test the final version of Verdant, Verdant-4. These two sessions, occurring on different days separated by 10–14 days in between, allow us to simulate the use of history in longer term projects.

Session 1 (S1). A data scientist does *exploratory programming* with Verdant available in Jupyter Lab. This coding session *generates history data*. Although we expect history data to be most helpful *later on after* a data scientist completes their initial analysis, we are looking out for any naturally occurring use cases where the data scientist pauses to consult history during coding.

Session 2 (S2). Between 10–14 days following Session 1, a data scientist is asked history questions about their analysis from S1. This is an opportunity to see how easy or difficult it is for data scientists to answer history questions about their own work using Verdant. We also aim to learn more about when, why, and how data scientists might use history data in real life settings through interview questions.

REMOTE STUDY SETTING

Due to the Covid-19 pandemic, this study needed to be fully remote. We conducted the study over Zoom, with our own server running JupyterLab with Verdant. All study materials were provided to participants via web links, so that the entire study could be conducted in a web browser. To record data, we gained permission from each participant to record an audio/video screenshare of their browser window while they worked on study materials.

PARTICIPATION

Using social media and our personal social networks, we recruited 11 participants (see Table 13.1) with a minimum of 1 year experience with each of: data science, Python programming, and Jupyter notebooks. Genders represented were 6 male, 5 female with a mean age of 25 (SD = 3 years). We sampled for a variety of data science expertise levels: five participants were data science professionals, while six were upper-level students. No two participants came from the same organization.

Table 13.1 Summary of Participants

	Profession	Data Science Years	Python Years	Notebook Usage	Freq of data work
E01	Data Scientist	2	6	Daily	Daily
E02	Postdoctoral Researcher	12	7	Occasionally	A few time a week
E03	Graduate Student in HCI	7	9	A few time a week	A few times a month
E04	Undergraduate in Computer Science	1	5	A few times a month	A few time a week
E05	Graduate Student in Electrical and Computer engineering	0.5	2	A few time a week	Daily
E06	Graduate Student in Astronomy	3	4	Daily	Daily
E07	Undergraduate in Computer Science	2	3	Daily	A few time a week
E08	Senior Business Analyst	2	6	A few time a week	Daily
E09	Graduate Student in Computer Science	3	5	A few times a month	A few time a week
E10	Research Software Engineer	2	3	A few time a week	A few time a week
E11	Quantitative Strategist	8	5	Daily	Daily

SESSION 1 PROTOCOL

S1 was 2.5 hours, with 30 minutes for study setup, tutorial, and interview, and 2 hours for the programming task. This is the maximum time commitment we felt able to recruit for, since our professional participants scheduled with us for after-work evening hours and weekends.

Session 1 occurred as follows:

1. (5 min) Introduce the study and verify consent. Then help the participant setup screen-sharing to their web browser, verify permission to record the Zoom call.
2. (13 min) Verdant Tutorial (<https://marybethkery.com/Verdant/tutorial/tutorial.html>)
3. (2 min) Interview break #1
4. (5 min) Introduce the programming tasks
5. (120 min or until 10 minutes remaining) Programming session
6. (10 min) Interview break #2 and schedule Session 2

We wanted to see if there are naturally occurring use cases where a data scientist pauses to consult history during coding. To set up that scenario, we first have the participant do an interactive tutorial where they use Verdant to uncover history facts from a pre-made notebook and pre-made history — much like what we did in our initial Jupytercon Verdant study (see Chapter 11). After completing the tutorial, the participant was free to consult the tutorial at any point during the study as reference for Verdant’s features. After completing the tutorial, we asked each participant:

Interview break #1: *From what you’ve seen of this kind of history functionality, can you see yourself finding this useful in your own workflow?*

This question allowed us to gain participants’ initial impressions of how appealing the tool’s idea was to them. During their actual programming session, we ask all participants to have Verdant’s UI open as they work, but did not require them to interact with it:

If you could have the tool open, that will help us monitor that data is being collected. If you should feel any reason to use the history functionality while you work, definitely do. But if you don’t, that’s fine too.

The experimenter remained silent and on-mute during the programming session, unless a bug occurred in Verdant where it was necessary to stop the participant: for instance to ask them to refresh the webpage to get Verdant working again. Otherwise, we let the participant work as they would normally and did not interrupt them. If we observed participants interacting with Verdant during the programming session, we took note of it to discuss in Interview break #2.

Finally, we ended programming tasks approximately 10 minutes before the end of the session, to wrap-up with **Interview break #2**. An open-ended conversation with participants started with:

How did you find the experience of having Verdant running to the side while you worked during this session?

If a participant interacted with Verdant during the programming session, we prompted them to describe what they were aiming to do in those moments, and whether they felt able to achieve their intended goal with the tool.

CHOOSING PROGRAMMING TASKS & DATA

In this section we go through the design choices that went into our task design for the S1 programming session. For Session 1 our ideal programming task would:

- *Fit a range of skill levels.* Be something a novice data scientist can make reasonable progress on within 2 hours, *but also* something that a senior data scientist will not run out of things to do within 2 hours.
- *Include plenty of exploration.* Ensure a high likelihood of seeing exploratory programming during the session.

Since exploratory programming is what we want to observe, we first narrow down to specific types of data science work that require heavy exploration. Exploratory data analysis (EDA) and *initial* machine learning model development are well documented to be exploratory in nature [Tukey 1977][Wongsuphasawat 2019]. That is as close as we can get to a theoretical guarantee within a user study that we *will see* exploratory programming if participants engage in EDA and model development as defined. The difficulty level and time expectation of EDA and model development, as any instructor who has written a homework for it can tell you, comes down to the dataset [Kandel et al. 2011]. Here I rely on my experience creating homeworks to create study tasks and choose appropriate data.

Choosing a Dataset

For our study, we need to choose (or create) a dataset that is “easy” enough that our participants will be able to do plenty of exploration without getting caught in time-expensive debugging or bug-related traps. Bugs will happen, but debugging is not an interesting behavior for our research questions, so we want to minimize it. An “easy” dataset will be clean, tabular, have relatively few columns, self-explanatory variables, and clear correlations between variables. The easiest of datasets for study purposes would be a standard “toy dataset”, such as the often-used titanic dataset¹ or cars dataset². However, there is such a thing as too easy a dataset. We need to choose a dataset that is “hard” enough that a data scientist of unknown skill-level will not run out of things to do within the few hours of a study. A “hard” dataset, like the sort a data science professional sees in real life, will be messy, have strange formatting, hundreds of columns, multiple data sources, unclear relationships between variables, hard to interpret variables, and so on [Won et al. 2003, Kandel et al. 2011]. The ideal study dataset will

¹ Titanic survival dataset 1999 <https://www.openml.org/d/40945>

² Cars make and model dataset 1985 <https://archive.ics.uci.edu/ml/datasets/automobile>

be just challenging enough to keep participants continuously busy exploring with minimal debugging.

To choose a dataset, it is often recommended to pick a standard published dataset. Creating your own dataset for data analysis study is risky, because it leaves room for doubt that perhaps you somehow (even accidentally) constructed your dataset in such a way that will bias participants' behavior. A standard dataset that is publicly available and has been used by others has a demonstrated range of things a data analyst can do with it. In practice, however, we found that most common standard toy datasets, like the titanic, iris³, census⁴, or cars datasets, were not suitable for a data science programming study. Toy datasets tend to be too “easy” to do much with, and you run the risk that some data scientist participants will have already encountered these toy datasets in common educational materials. A situation where a participant has already analyzed your chosen dataset in the past might invalidate that participant entirely from being a valid research subject.

My current favorite place to find datasets suitable for both homework assignments and studies is Kaggle (also used in Jupytercon Scavenger Hunt Study, Chapter 11). Besides the competitions, Kaggle is a place where individuals or organizations post datasets and then data scientists post notebook analyses of those datasets. For our purposes, Kaggle has all the benefits of a standard dataset in that by browsing the publicly available notebooks that data scientists of a range of skills have posted for that data set, we can still preview the range of analyses and modeling that is possible for that dataset. Like other public dataset repositories, Kaggle has a catalogue of datasets searchable by licensing, topics, popularity, and so on. Finding a dataset ideal for your study does take a bit of digging. For this study I searched for a dataset that was:

- **Tabular.** This is important for our study because although some data scientists will be more familiar than others with other kinds of datasets like text, pictures, sensors, etc., we can expect all data scientists to be familiar with tabular data since it is what is first taught. This is a way of controlling for expertise across participants, since we don't want to pick a data format where some participants will approach the data very differently than others due to their expertise.
- **Appropriate licensing** that allows academic use and appearance in publications.
- **Small enough to quickly compute locally** on any commodity computer but large enough to allow for splitting the dataset for machine learning and different ways of slicing the data for analysis. As a rough heuristic, I looked for 10-20 columns and tens or hundreds of thousands of rows, under 2GB. Like debugging, processing time is time taken away from the actual kinds of behaviors we hope to observe for research purposes, so an overly large dataset can undermine a programming study.
- **Approachable topic matter.** An everyday topic like food, weather, houses, or cars we can expect all participants to roughly have the same expertise with. Again, specialty topics

³ Iris classification dataset 1936 <https://archive.ics.uci.edu/ml/datasets/iris>

⁴ Census income dataset 1994 <http://archive.ics.uci.edu/ml/datasets/Census+Income>

like finance or Pokémon™ can cause issues if some participants have a lot more domain knowledge than others, since they will know to do certain things with that data that those without domain knowledge will not.

- **Self-explanatory columns.** Column names that mean something obvious, such as “product name” or “zipcode” will be easier for a data scientist to quickly start working with in a study.
- **A variety of insights that take a bit of work.** In looking at the publicly posted analyses accompanying a dataset you can see how much code or work it takes for a person to reach a particular insight. In looking across analyses for a dataset, you can get a sense for the range of insights people reach about that data. An example of a bad fit for our study is the Titanic dataset. After a few lines of code most analyses conclude class and gender are the features that determine survival on the Titanic. Subsequent analyses can refine indicators in the dataset of class and gender, but there’s really only so much you can do with the dataset. The amount and variety of work in the posted analyses is a kind of preview of the amount and variety of work you may see from participants in your study.
- **Amenable to common Python libraries.** This generally comes for free. If you have already found small-ish tabular data, it will almost always be amenable to today’s standard libraries like pandas or scikit-learn. Avoid datasets where most analyses use a specialized library or tool, because again, learning a new library can distract significant time away from the behaviors we’re hoping to observe during the study session.

From this search we chose a dataset of Kickstarter crowdfunding projects collected by Mickaël Mouillé and available at <https://www.kaggle.com/kemical/kickstarter-projects>. The Kickstarter data is available under a Creative Commons license and is tabular, with a nice variety of nominal, categorical, date, and textual columns within a small set of just 15 columns. 13 of 15 columns represent real-world concepts like “title” or “country” that are easily explained. The variety of data types among the 15 columns/features leave room for interesting feature creation. The data has 1 unique row per Kickstarter project, and a single column “state” that contains the variable for a model to predict. This means that the data is already in a format that can be easily inputted into a machine learning model as-is. With 375,765 rows, the dataset has enough data to work with most models and analyses, but since the data is all tabular, the whole dataset is under 56mb, which is small enough to quickly process on a commodity personal computer. Finally, our examination of the publicly available analyses for this dataset confirmed that there are a variety of insights people can discover about Kickstarter data, and that there are no models that can predict Kickstarter project success with high certainty. This means that we do not anticipate any data worker to be “done” with the dataset under 2 hours of our programming session.

The Programming Task

The full prompt given to participants is shown below:

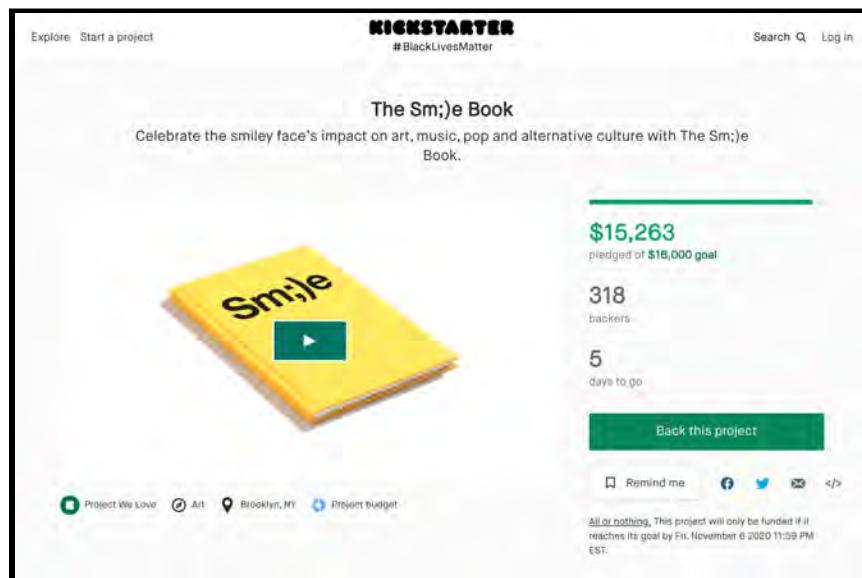
Verdant Usage Study: Session 1 Tasks

Goal Overview

Kickstarter is an online platform where people contribute money to *crowdfund* a project such as a book, game, or music. With this startup funding from the crowd, the project team is able to produce their idea as a real-world product.

You are given a dataset of over 300,000 Kickstarter projects. As Kickstarter explains it:

"Every project creator sets their project's funding goal and deadline. If people like the project, they can pledge money to make it happen. If the project succeeds in reaching its funding goal, all backers' credit cards are charged when time expires. Funding on Kickstarter is all-or-nothing. If the project falls short of its funding goal, no one is charged."



Example of a typical Kickstarter project. Image Source: Kickstarter, screenshot of <https://www.kickstarter.com/projects/thesmilebook/the-sm-e-book>

Client: You are being hired by a business school professor who wants to teach students about crowdfunding in their "Internet Entrepreneurship" course where students will create their own Kickstarter startup pitches for a class project.

Goals:

1. Create an exploratory data analysis that can inform your client about what makes a *successful* Kickstarter project. A successful crowdfunding project reaches its complete financial goal.
2. Your client would also like you to communicate any interesting patterns about what might make a Kickstarter project *fail*.
3. Your client wants to know if you can build a machine learning model such that a student can input data about *their* project idea and your model will output how successful that project is likely to be. However, they would also accept if Kickstarter projects are not that easy to predict. The professor wants you to tell them if such a model is actually impractical.

Packages: To install a missing package, type and run **%pip install <package>** in a cell in your active notebook

Deliverables: In Session 2 (roughly a week from this study session 1), you will be asked to present & discuss your work for this client in a Jupyter notebook format.

Note that this study is not evaluating your level of coding, statistics, or machine learning skills. Please work as you normally would.

Note the dataset for this study comes from [kaggle.com](https://www.kaggle.com). You are free to use the internet to help you code, just **do not go to kaggle.com** or any place you believe will directly give you the "answers" for the client's requests about this dataset

Figure 13.1 Session 1 Programming Prompt

To help control for domain expertise among participants, the data is introduced with a short summary of crowdfunding and Kickstarter for participants. We invent a fictional client (rather than ourselves being the client) to help position the experimenter as a neutral observer. If participants attempt to refine or discuss the client's goals e.g., "So did they mean X?" the experimenter replies that they don't know what the client meant. This is to ensure all participants receive exactly the same task information. We also kept the goals of the client somewhat open-ended and high-level, consistent with the common characterization of client requests of data scientists [Passi & Jackson 2018, Hou & Wang 2017].

SESSION 2 PROTOCOL

Session 2 occurred between 10–14 days after S1 in our attempt to replicate a key use case for experiment history where a data scientist is asked about or needs to return to past work. Our primary goal was to test how easy or difficult it is for data scientists to answer history questions using Verdant. The setup is very similar to both our Session 1 tutorial and our earlier usability test with Verdant at Jupytercon 2018 (see Chapter 11) in that we design a “scavenger hunt” for each participant to answer questions about their analysis using history.

Session 2 occurred as follows:

1. (5 min) **Setup:** Introduce the study and verify consent. Then help the participant setup screen-sharing to their web browser, verify permission to record the Zoom call.
2. (10 min) **Recall:** each participant starts S2 by looking over their notebook from S1 for up to 3 minutes. When they are ready, participants are asked to go top to bottom in their notebook and describe out-loud what they did during S1. This recall step is meant to mimic the time a data scientist might have to briefly skim over their work before a meeting. This recall step is also designed to help control for individual differences among participants with different base memory abilities, by giving all participants the same refresher time before beginning the scavenger hunt.
3. (20 min) **Scavenger Hunt:** each participant is given the tutorial webpage from S1 to use as reference. Next, each participant is given their scavenger hunt questions one at a time pasted as text into Zoom’s chat feature. Participants are asked to answer the questions out-loud. The experimenter is silent, intervening only in case of a bug with Verdant or JupyterLab.
4. (15 min) **Interview:** After the scavenger hunt, participants are interviewed about their opinions of Verdant as well as their normal data science and data science history practices.
5. (5 min) **Debrief:** Finally, participants are given a few minutes at the end to ask any questions they have about the study, the tool, and the research itself.

Following Session 2, participants were compensated \$20 for their time.

GENERATING SCAVENGER HUNT QUESTIONS

Given that S1 involves exploratory programming for exploratory analysis and model development, we knew that each participant would develop their analysis and ideas in different ways, requiring us to personalize the history questions we ask each participant. This presents a major study design challenge, especially since we still want to compare question answering *between* participants. Thus, although each participant is given a unique set of questions, we created a detailed question-generation protocol to help us create as consistent question sets as possible across all participants. This includes first labeling the participant’s history and then following the question generation criteria, as discussed next.

Prep work: Labeling Programming Session Activity

Following S1, we took each participant's history data generated by Verdant. I used Verdant's Ghost Notebook feature to look through each version of a participant's session, one at a time, and manually annotate what happened in that version. Besides labeling each version with a brief description, for each version, I assigned a category of activity based on changes the participant had made in the version. The category set, shown in Table 13.2, emerged through open coding, drawn from common data science activity types.

Table 13.2 Scavenger Hunt Task Categories

Data Cleaning & Filtering	Modeling & model pipeline	Feature creation & selection	Tables & Summary Stats	Visualization	Storytelling & Notebook Organization

Note that due to the quick turn-around time needed to do all data labeling between sessions, we did *not* do a collaborative coding approach, where 2 or more people separately label the data, typically done in thematic analysis for reliability [Charmaz 2006]. I labeled all data myself for consistency, such that my personal biases in deciding labels affected all participants evenly. Due to the iterative nature of coding, codes shifted a bit during the initial 3 participants of the study E01, E02, and E03. For instance, "Data Cleaning & Filtering" was originally combined with the category "Tables & Summary Stats". By E03 I had coded enough data that the category set was finalized for the rest of the study. I then re-coded the first 3 participants' activity to ensure their activity labels were consistent with the final category set.

An example of this version coding is shown below. Where a change could belong in multiple categories from Table 13.2, I conservatively assigned one or two categories that fit best to what *resulted* in a version. So, for instance, in the example below, the participant removes a plot that had been in version 47, but since there is no plot in version 48, I did not assign the "Visualization" category. I just assigned the "Tables & Summary Stats" category since the *result* of the participant's change is a new table showing, replacing the plot.

Table 13.3 Example of a version categorization & description from S1 activity

Version #	Timestamp	Activity Category	Description
48	2021-03-22 18:45:55	Tables & Summary Stats	erases plot and shows merged in table instead

Prep work: Generating Realistic Scavenger Hunt Questions

What makes for a realistic Scavenger Hunt Question? A realistic question ought to be one that a boss, team member, paper reviewer, or the data worker themselves would reasonably ask about the data worker's analysis and modeling from S1. To guide realism, we grounded questions in the

history queries we obtained from practitioners in the Query Design Exercise (Chapter 4). I also leveraged my own domain experience with data science and as a teacher of data science course materials. Finally, as an intervention check, we *asked* participants during the interview portion of S2 to discuss how realistic the questions were, and how often they got questions like that in their real life practice. Participants largely affirmed the realism of the scavenger hunt questions, which give us higher confidence in our study results. Participant feedback also gave us important clues for improving the study protocol. For instance, we learned from our first two participants, E01 and E02, that questions about something they **did not do** in their analysis was actually an important kind of question they are asked in real life. From this feedback we added a rule to our question generation procedure below, to ensure all participants are asked this type of question.

To generate scavenger hunt questions for each participant:

1. For each of the 6 activity categories (Table 13.2 above) I generated 1 question pulled from participants' versions that fit that category. This creates **6** questions, with a minimum of **4** questions because in some cases, a participant did not do anything for a specific category. For instance, several participants ran out of time before they did anything related to modeling, and thus had no versions labeled with “Modeling & model pipeline”.
2. We included 1 question (of any category) about something the participant **did not do** in their analysis, but could have plausibly done given their analysis history.
3. We structured each question in the following way to help ensure consistency between participants:
 - a. First, each question has a “**find**” clause, asking the participant to locate a specific thing in history.
 - b. Second, each question has an “**explain**” clause, asking the participant to explain something about the artifact they were asked to find.

Question examples below demonstrate the overall flavor of the question sets:

E10 Q1: (Visualization category)

- a. **Find:** Go back to when you had a plot for comics
- b. **Explain:** Are there any categories of comics that are substantially more successful than others?

E08 Q3 (Feature Creation & Selection category)

- a. **Find/Explain:** What different features did you try in the model?

E04 Q8: (Feature Creation & Selection category)

- a. **Find:** What kind of performance change did you see in including the smaller categories feature in your model?
- b. **Explain:** What is your intuition about how helpful the categories feature is?

E06 Q2 (Table & Summary Stats category, something they did **NOT** do)

- a. **Find/Explain:** Having no country as Country N,0" seems to be highly predictive of a project failing. Did you ever do anything with Country N,0" or filter those projects out of the dataset? Why or why not?

In total, we originally planned for each participant to do 10 questions, however in practice we could not fit 10 questions into the planned time. Most participants completed 6 questions, although one participant E11 completed only 4 questions in the given time period.

The question order was then shuffled using Python's random.shuffle. Finally, we took the first 3 questions and assigned them to a specific feature the participant would start their search at, between Artifact Summary, Search, and Inspector. Since the first 3 questions were random, we did *not* assign the start feature randomly, but rather matched the 3 questions so that there were one-each of Artifact Summary, Search, and Inspector based on which feature made the most sense for each question.

From the 4th question til the rest, the participant was told that they could use any features in Verdant. The goal of assigning participants to start at specific features for the first few questions was to ensure they were exposed to all features of the tool enough to know about them. Note that we did not assign participants to start at the Activity Pane because we did not ask any time-based questions. The reason we did not ask time-based questions (e.g., “what was your model result at 4pm last Thursday?”) is that we only had a single two hour session of work per participant. With only one small chunk of time to refer to, I believed asking participants to return to a specific timestamp would be unrealistic. However, although Activity Pane was not assigned for S2, note that all participants were asked to have the Activity Pane open during their work in S1, so the feature did see usage during the S1 coding session.

PROTOCOL EVOLUTION

During the study we did encounter some circumstances that led us to adjust the protocol or Verdant.

Since Verdant is a prototype tool, we did encounter bugs during the study that we needed to repair mid-study. These caused some data loss or corrupted data. Most critically, participant 5 did not continue on to S2 due a data collection failure during S1. Meanwhile participant 6 and participant 7 had corrupted data such that their history data from S1, while otherwise valid, has incorrect timestamps. Thus participants 5, 6, and 7 will be excluded from some (but not all) of the analyses below in cases where their data is not fit to be analyzed. Additionally, since Verdant needed to be fixed mid-study, there are some usability issues arising from software bugs that affected participants early in the study, but not those later in the study. These are discussed in our usability findings. Overall, we did **not** include a standard usability survey, like SUS, to

compare across participants, because participants early, mid, and late in the study experienced substantially different levels of bugginess impacting the usability of Verdant.

Second, we discovered through recruitment that some participants are *too junior* for our study protocol. While every participant we recruited was familiar with data science, programming, Python, and Jupyter Notebooks, we found that some participants demonstrated less practical experience with doing Python data science. These more novice participants spend considerable time debugging library calls or figuring out how to do routine data transformations, which for our logs resulted in more debugging than exploration. To ask realistic scavenger hunt questions for S2, we need participants to do enough exploration in 2 hours for us to ask at least 5 different history questions about. For two participants, their logs did not contain enough different analysis, models, or visualizations to ask about, leading us to come up with questions that sound a lot more like something a course instructor or TA might ask:

E07 Q2 (Data Cleaning & Filtering category)

- a. **Find/Explain:** There's df2, df2_onehot, and onehot in the notebook, which are all variables that appear to have to do with onehot encoding but don't end up in the model later in the notebook. Please use history to explain what these were used for.

We still hold that this type of question is valid, since a student answering questions about their work is one of the usage scenarios in Chapter 10 that we designed Verdant to support. By midway through the study we put more work into our recruitment efforts to recruit more experienced practitioners.

Finally, the first three participants of S2 did not find the Inspector feature on their own – even when prompted to use it for the scavenger hunt and provided with the tutorial that describes it. This lack of discoverability for the Inspector led us to explicitly intervene for the rest of the participants, and verify that participants could identify the Inspector tool before starting the scavenger hunt.

LIMITATIONS

This study has several key limitations, some of which are limitations of the study design and some of which are results of unforeseen mishaps that occurred during the study.

First, we do *not* include a control condition in this study design. Since much of our work in the Exploratory Programming Study (Chapter 3) and the Notebook Usage Study (Chapter 4) has already examined how data workers work in the absence of good history support, we did not believe that having programmers follow the S1 and S2 protocol *without* history support would be a helpful exercise. One reasonable control condition might be to have participants answer the scavenger hunt questions from S2 using the versions collected by Verdant as files (`notebook_v1.ipynb`, `notebook_v2.ipynb`, `notebook_v3.ipynb`, ...). However, on average

participants in this study generated 135 versions, and we have good reason to believe from our prior studies and others' [Ragavan et al. 2016] that manually searching through 135 files for answers would be time consuming and unpleasant for participants. For instance Ragavan et al. in their 2016 study had participants do a series of just 3 coding tasks over 700 file versions, and participants spent the vast majority of their 50 minutes foraging through files looking for the right information. The downside of omitting a control condition is that we cannot say Verdant is measurably better or faster than a baseline. **This study is not a controlled evaluation study.** Instead, at this stage of research we are conducting an exploratory study to understand the observable benefits and pitfalls of our proposed designs for making history easier to use. This is the first time data scientists are conducting exploratory programming and answering questions about their own history using Verdant. Thus our design includes a mix of programming, interviews, and structured history tasks to gain rich qualitative and quantitative data on not just how Verdant performs, but how Verdant *performs and fits into a data scientist's existing workflows and practices*.

In all, 11 data scientists were recruited for the study. Although we attempted to recruit a representative sample of participants and construct tasks representative of common real-world data science programming, there is no guarantee that results from this small exploratory study will generalize. Since participants volunteered to participate in a study about history-keeping for data science, our participants may have higher than average interest in history tools.

CHAPTER CONCLUSIONS

Since the Verdant Study is a somewhat elaborate setup to rigorously simulate real exploratory programming data work, it did not come as a surprise that we experienced some study hiccups, data loss, and software bugs. The data we collected still meets our goals of proving a measurement of Verdant's performance with high external validity. With an emphasis on realism and rich qualitative data on top of quantitative measures, we have data to develop our understanding of how effective history support could serve data work practices in real life. The next chapter includes the results.

Chapter 14: Results & Discussion

Analysis done in collaboration with Brad A. Myers and Xinyi Zheng

INTRODUCTION

This chapter will be broken down into first quantitative results from session 1 and 2 tasks, and then qualitative discussion of our interview and observation data. For specific analysis, we detail each result on how it was analyzed and achieved. Participants will be referred to as E01, E02... E11. Sessions 1 and 2 will be abbreviated as S1 and S2 respectively. Overall, both quantitative and qualitative results are highly promising in support of Verdant as an effective addition to an exploratory programming data workflow.

PARTICIPANTS ARE FAST TO ANSWER HISTORY QUESTIONS USING VERDANT

For each participant, we labeled the S2 audio and video data with the timestamp where each scavenger hunt question starts and ends. Recall that each scavenger hunt question has two parts: a **find** clause and an **explain** clause, so we also labeled where each part starts and ends. The **find** clause ends when the participant reaches the specific historical evidence needed to answer the question – or else concludes that no such evidence exists. In most cases participants found historical evidence *and then* explained it. However in a few cases, especially when participants are searching for something they, in fact, **did not do**, explanation and search completely overlap as the participant searches and discusses at the same time, until they finally give up the search and conclude that no historical evidence exists. How long participants took per question is summarized in Table 14.1.

To determine success on scavenger hunt questions, we marked a question as **successful** if the participant found specific historical evidence (a specific artifact and version) that appropriately answers the question, or else correctly concluded that no such evidence exists. If a participant pointed out something not specific enough, like vaguely referencing all the history results from the search pane, we prompted them to continue on to find a specific artifact/version. Note that we only graded the success of the **find** part of questions. We chose not to grade participants' explanations. For the purposes of this study, we are not attempting to grade participants' data science skill or ability to effectively explain their work, although this could be done in future follow-on analyses of the data.

Every participant was successful with every question they attempted, except for 1 case where a participant gave up on 1 question.

Table 14.1 Scavenger Hunt Question Completion Times

Question Type	Question Count	Average Time to Answer (in min:sec)
Find	64	1:26 (SD=1:01)
Explain*	64	1:02 (SD=0:40)
Data Cleaning & Filtering	9	1:12 (SD=0:34)
Feature creation/selection	17	1:26 (SD=1:02)
Modeling	12	1:28 (SD=1:01)
Storytelling & Notebook Organization	4	1:37 (SD=1:42)
Tables & Summary Stats	9	1:12 (SD=0:59)
Visualization	13	1:40 (SD=1:08)

Of the 64 scavenger hunt questions attempted across all participants, 60 were successfully answered using Verdant. For those *not* answered using Verdant, 3 questions were simply answered by participants from memory without consulting Verdant, and for 1 question the participant gave up on finding an answer in Verdant after it was taking too long to find. This is a success rate overall of **98%**.

Timing by if/how a question was answered is shown below:

Table 14.2 Scavenger Hunt Question Completion Times By Answer

Answer	Question Count	Average Time to Answer
Successful	63	1:23 (SD=0:58)
Failed	1	4:03
Successful - Verdant	60	1:25 (SD=0:58)
Successful - from Memory	3	0:49 (SD=0:38)

How do these results compare with our prior Jupytercon Scavenger Hunt Study?

To get a sense of how these results compare, we would like to compare against how participants did in the Jupytercon Scavenger Hunt Study. There's a clear limitation to this comparison: participants in the Jupytercon hunt answered questions about *someone else's* analysis history, while participants in this hunt answered questions about their own history. So, we won't be able to separate the effect of our UI changes from the effect of memory on participants' performance. Nonetheless, the comparison may be a helpful benchmark.

To make a valid comparison, we needed to do a bit of re-analysis of our Jupytercon Scavenger Hunt Study data because the study described in Chapter 10 originally used different measures of "success" and also tested participants on scavenger hunt questions that are different in nature from the scavenger hunt questions in this study. To make the two studies comparable:

1. We retrieved the *time* it took participants to answer each question. This timing was recorded but not reported in the original Jupytercon study analysis.
2. We took a *subset* of 10 of the original 13 scavenger hunt questions from the Jupytercon study that, like our scavenger hunt questions in this study, were *realistic* history analysis questions that might be asked in a real work scenario. This meant throwing out the 3 questions A, B, and C from the study which had been designed as *artificial* questions purely to test UI elements. The discarded questions are:
 - A. Find the first version of the notebook
 - B. How many cells have been deleted
 - C. How many runs did the author leave a comment on

To ground in realism, the remaining 10 questions from the Jupytercon study were based on real history queries by data scientists in the Query Design Exercise (Chapter 4). The 10 questions match the format of the "find" type questions in this current study. Since participants in the Jupytercon study were looking at *someone else's* work, we did not have any "explain" follow-up questions like in this study.

After this reanalysis, participants in the Jupytercon Scavenger Hunt Study have 44 of 57 tasks successful for a 77% total success rate. Note that in Chapter 10 we reported 60 of 81 tasks successful for a 74% total success rate, so this is not a major shift despite the change in analysis approach. Interestingly, just as our single failure case in the current study was from a participant giving up, **giving up out of frustration accounted for 9 of the 13 failure cases** in the Jupytercon Scavenger Hunt. For us, this lends evidence once again reinforcing the idea we've seen throughout this dissertation (Chapter 3, Chapter 10) that when retrieving history facts is difficult to do, data workers simply won't do it.

For timing, we found that participants in the Jupytercon Scavenger Hunt completed questions in an average of 2:40 (SD=2:10). In contrast, participants in this study successfully completed 63 of 64 finding tasks for a 98% success rate in an average time of 1:26 with a much lower standard deviation of 1:01. This is a major improvement.

EVERYONE TAKES A DIFFERENT PATH IN EXPLORATORY PROGRAMMING

Verdant's logs allow us to analyze how S1 programming sessions unfolded.

Recall that for S2, we manually assigned each version in a programming session to one of 6 possible activity types (Table 14.1). Below (Figure 14.1), we plot each participant's versions on a timeline by activity type. Timelines for each participant are shorter or longer depending on how many minutes they spent programming. This chart allows us to visualize the sheer variety of approaches participants took on the same data and same tasks. For instance, we observe that E04, E10, and E11 are particularly meticulous about tidying and writing detailed markdown in their notebook, as indicated by the large proportion of brown bars in their timelines. We see that E05 and E08 favor visualizing their data early on to get an overview, as indicated by blue bars. E02, E03, E10, and E11 prefer to start by looking at tables, as indicated by green bars at the start of their sessions.

Although we can spot some similar activity patterns between participants, the degree to which these timelines *do not* look alike, given the same dataset and same prompt, is an interesting indication of how personal exploration is.

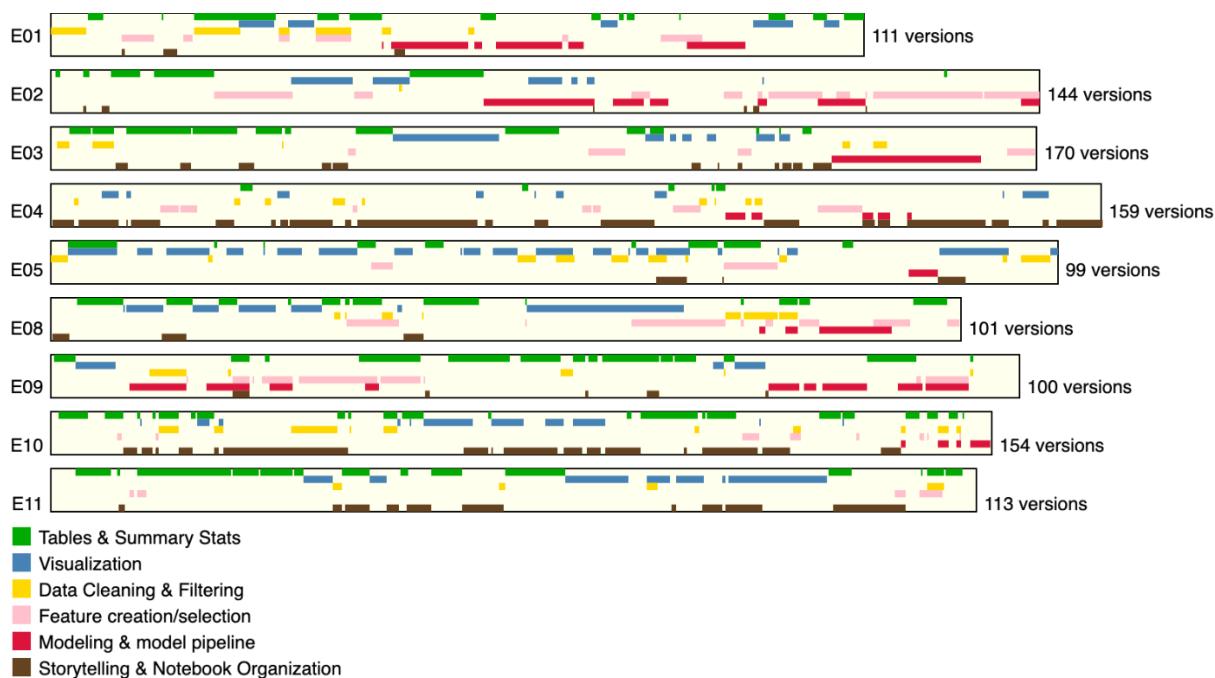


Figure 14.1 Colored blocks show the type of data science activity each participant engaged in over the duration of the programming session. The length of rows vary based on the total time of each participant's programming session.

Next, we compared version counts across participants (Figure 14.2). Although we planned for 2 hour coding sessions, intro materials often took longer than expected, and participants worked for 100 min on average ($SD=7$ min). Only 1 participant ever declared themselves done with all analysis tasks, and all other participants ran out of time as expected. Participants generated an average 135 versions ($SD=33$ versions) or roughly 1 version per minute. We did not find any correlations between a participant's years of experience with data science and the total number of versions they generated.

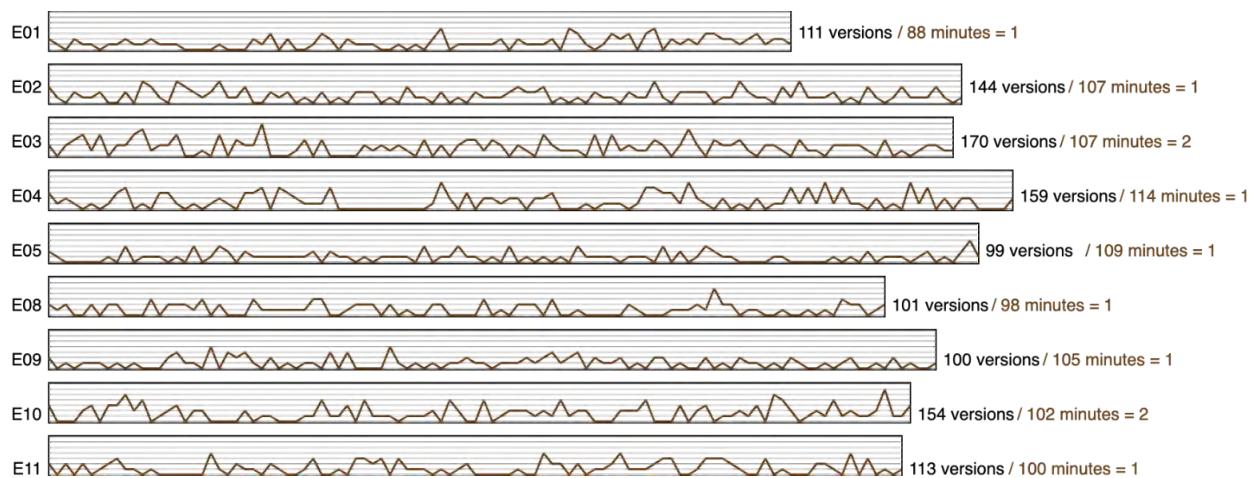


Figure 14.2 Rows show the rate of how many versions are recorded per minute over each participant's coding session, with the total number and average per minute labeled at the end of each row. Participants 6 and 7 are excluded due to timestamp errors in their log data, as discussed above.

PARTICIPANT EXPERIENCE USING VERDANT

We have positive evidence to suggest that practitioners engaged in substantial exploratory programming with Verdant, and were overall successful in later recalling the history of their work using Verdant. Next, we use a combination of log data and qualitative analysis to dig deeper into the experience of using Verdant, and how participants perceived Verdant's form of history support.

Qualitative Thematic Analysis

For all interview data in this study, we conducted a thematic analysis [Braun & Clarke 2012, Charmaz 2011]. First, we transcribed all interview audio from S1 and S2 for each participant. To prepare transcripts for qualitative coding, I segmented the transcript for each participant into

small chunks of 1-10 sentences that share the same topic. This segmentation was organized in a spreadsheet for each participant. This initial segmentation would later be refined during the coding process, as we chose which sentences grouped best together in a participant's speech. Next, as the researcher most familiar with the data since I was present at all study sessions, I conducted the first stage of open-coding on the data to create our initial code book. Another researcher, Xinyi Zheng, then did her own round of open-coding on a sample of the data, using these initial codes and adding her own. Each participant quote was allowed multiple codes if it touched upon multiple topics. Through a collaborative discussion, we then finalized the code book with 55 codes. Finally, both researchers split the data and each coded half of participants. With initial coding complete, both researchers then iterated groupings of 55 codes through discussion to decide on 11 axial code “themes” [Braun & Clarke 2012, Charmaz 2011]. All 55 codes are included in Appendix F.

All quotes reported have been trimmed and lightly edited for grammar and clarity.

Working alongside a history tool

During the programming session, we observed 3/10 participants pause while working to use Verdant to look up some history. At the end of the programming session, as an intervention check we asked all participants “*How was the experience of working alongside the history tool? Did you find yourself looking at it or just tuning it out?*” From these questions we found that 4/10 said they passively watched the activity visualizations updating as they worked, because it let them know that their work was being saved. For E03 it helped them appreciate the work they were doing: “*it just makes me in awe of how much I've typed*”. E02 used the activity view during programming to debug:

“I was bugging out down here. I looked at my tracker and realized that I hadn't actually edited one of the early code blocks, which meant that the pre-processing hadn't pushed itself through. Um, and so this was super, that was super helpful for me to kind of see, um, where I had been in the code.” – E02

For the rest of participants who did *not* interact with history, we heard from participants that this is largely because in a short initial programming session, users had a good memory of what they had done so far:

“I didn't rely much on the history to be honest, because this was just one sitting, but, if I were to revisit the code at periodic intervals, then I might have actually looked at my code and see where I stopped at... because that would allow me to make my future decisions on that basis. But yeah, because it was one sitting I didn't end up using much of the history tool.” – E07

Between the tutorial and programming session in S1, we asked all participants: *From what you've seen of this kind of history functionality, can you see yourself finding this useful in your own workflow?* 7/10 participants were enthusiastic about Verdant's usefulness, answering “*Oh my*

gosh, yes." [E03] or "*I can see it being extremely useful.*" [E09], while 3/10 participants had specific reservations about Verdant's style of history. For one participant, E08, their primary reservation was simply that they preferred R and RStudio, and couldn't see themselves working in Jupyter notebooks often enough to justify using Verdant. Meanwhile, E01 had reservations about cell-based history and E11 had concerns over automatic versioning. Later, after using Verdant for their own work for the S1 programming session and S2 scavenger hunt, E01 and E11 expressed more positive feedback: "*I think this is definitely a useful tool*" [E11]. However, for E11 and E08, Verdant still was mismatched to their preferred work practices. E11 primarily works with super computers and large-scale analyses, so they don't tend to use Jupyter for long periods of work where history would be as needed. E08 prefers data programming with R, and was also confident and content in their history practices as-is. E08 did acknowledge, however, that although they are satisfied with their own history practices without Verdant, they would appreciate Verdant when they need to understand data work code written by someone else. From beginning to end of the study, we did not observe any participants switch their beliefs from positive to negative on Verdant's usefulness after using Verdant.

All participants described specific use case needs for the history in their own experimentation work practices, shown in Table 14.3. The most popular need, expressed by 7/10 participants, was for easy collection and access to history of outputs, especially plots and images. This likely stood out to participants because Verdant includes full output histories linked to code but this is a rarity in today's version control tools.

Table 14.3 Participants' Use Cases for Verdant-Style Experiment History

Expressed use case for their own work	Participants
History of Outputs, Images, & Plots	E01, E02, E03, E04, E06, E09, E11
Model & model metric history	E02, E03, E04, E09
Describing history of data work to someone else	E01, E02, E10
Generally referring back to code history	E07, E09, E10
Refamiliarizing yourself with an older project	E02, E06
Using history to keep a more clean "current" notebook	E02, E08
Trying to understand the history of data work started by someone else	E08

The next most popular need, mentioned by 4/10 participants, is the ability to easily record and access model history. Since Verdant records every run of code, each run of the model under different conditions and parameters is all recorded. In real data work practice, Verdant's style of history keeping will likely be most handy for the early exploratory stages of model

development, because at later stages of development individuals or teams typically build a more formal logging pipeline for model experimentations.

Many of the other use cases for history were about communication. 3/10 participants talked about wanting to use history to describe the process of their data work to someone else – including answering other people's history questions like our S2 scavenger hunt questions. 3/10 participants talked about using history to reorient themselves to their own past work, and 1 participant talked about using history to orient themselves to data work started by a colleague.

Finally, 2/10 participants wanted history as an *alternative* to their personal practice of keeping all work in the notebook (see Chapter 4 for a discussion of this practice), which tended to result in cluttered notebooks. With the ability to rely on history, these participants felt that they would feel safe deleting old work from their current notebook, to keep a more curated notebook of just their end results (see Chapter 4 for a discussion of notebook curation). One of these participants in fact, later said that they *did* keep a more clean “current” notebook during the S1 programming task because they trusted history was being saved.

Cell-based history requires a certain workflow

6/10 participants brought up how the cell-based notebook history would either affect or conflict with their current workflow. This was surprising to us because we were not expecting trade-offs of cell based history (described in Chapter 10) to be so salient to first-time users. As E04 describes:

“You need to learn to keep all your similar things happening in one cell if you want to be able to trace it back.” – E04

Since many participants (6/10) worked in the expand-then-reduce pattern we observed in Chapter 4 where they created lots of small cells and later combined them, this meant that Verdant *wouldn't* work as well, since either history of a topic like “the model” would be distributed between lots of tiny cells or history would be lost when content snippets were moved and combined between cells. As discussed in Chapter 10, maintaining provenance of code snippets moved between cells is a challenge Verdant does not currently support. All history in Verdant can be retrieved through the search bar, but still, making history more flexible is an important area for future improvement. It was actually E04, who was very aware of this limitation, who was *also* the single participant to give up on a question: they found that the history they were looking for was scattered between multiple cell histories and decided that it was not worth their time to try to piece together code snippet history needed to answer the question.

What Features did Participants Use in the Scavenger Hunt?

To answer these questions, we analyzed S2 video and audio for each participant and labeled the *feature* of Verdant a participant is using for each second during a question. We determine a participant is using a feature by the following criteria: the user is actively typing, pointing,

clicking, or scrolling within that feature. Since the user's cursor can only be in one section of the screen doing one activity, we assigned only 1 feature per second based on the most precise feature we could say a user's cursor was touching.

Figure 14.3 below depicts the percentage of time out of the scavenger hunt that each participant spent using each feature. Note that some participants favor certain features (for instance E11 seems to spend most of their time in Artifact Detail while almost never using the Ghost Notebook), but overall some features are clearly more popular than others.

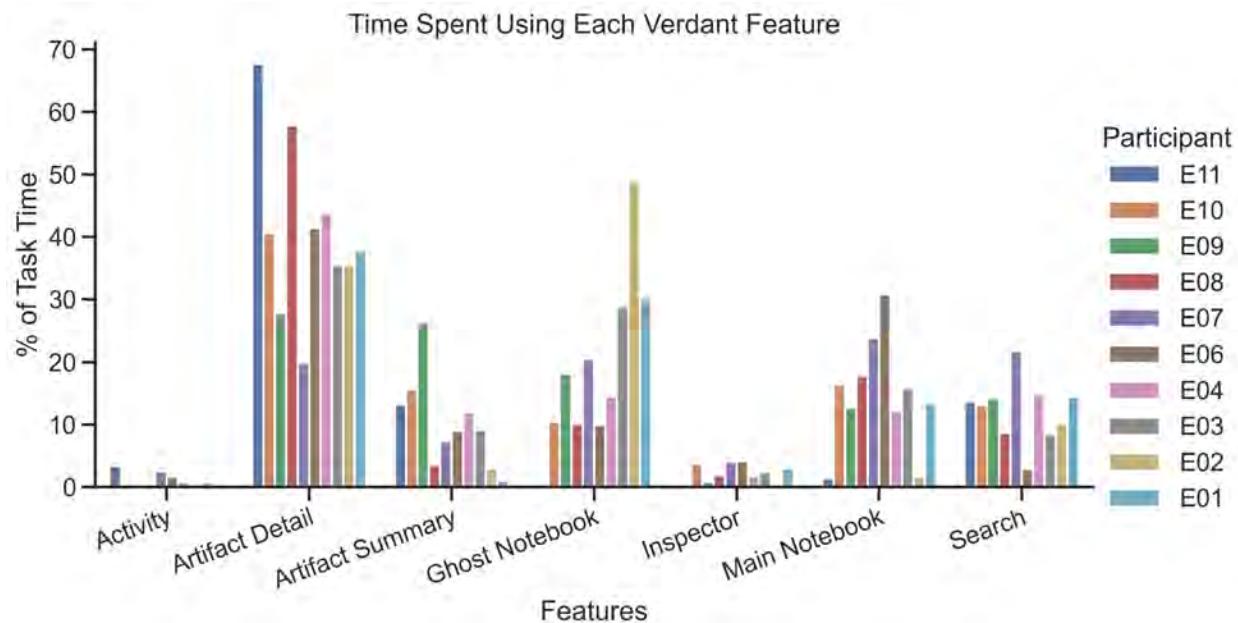


Figure 14.3 Percentage of scavenger hunt time that each participant spent using each feature. Main Notebook indicates time spent in the user's regular Jupyter Notebook, rather than in any of Verdant's history features.

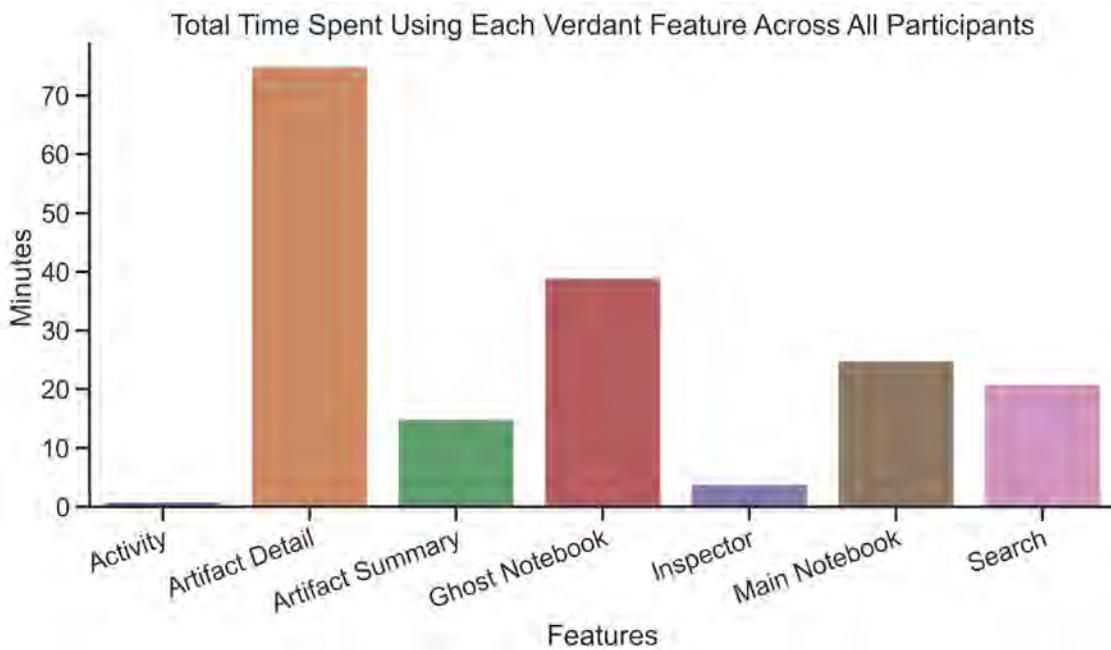


Figure 14.4 Cumulative number of minutes spent using each feature across all participants.

Overall, in Figure 14.4 we can see the total time spent in each feature is: #1 Artifact Detail, #2 Ghost Notebook, #3 Main Notebook, #4 History Search, #5 Artifact Summary, #6 Inspector, and #7 Activity Pane. Note that we just have timing here for feature usage during the S2 scavenger hunt, so this excludes, for instance, participants' usage of the Activity view during S1. This same ranking holds if we consider feature usage in terms of *number of times visited* instead of time spent at each feature. A participant visits a feature if, from a different feature or the question starting point, they switch to using that feature. Considering visits allows us to dig into feature usage further by characterizing the pathways users take through Verdant and their notebook to answer history questions. Below, feature node size indicates the number of times that feature was visited, across all participants. The arrows below indicate a user going from one feature to another, where the darkness and boldness of the arrow indicates the number of times that path was taken across all participants. Thus the bold paths and the biggest nodes show the most common feature usage across all participants. The [Start] meta node indicates which features participants visited to start their search for each scavenger hunt question.

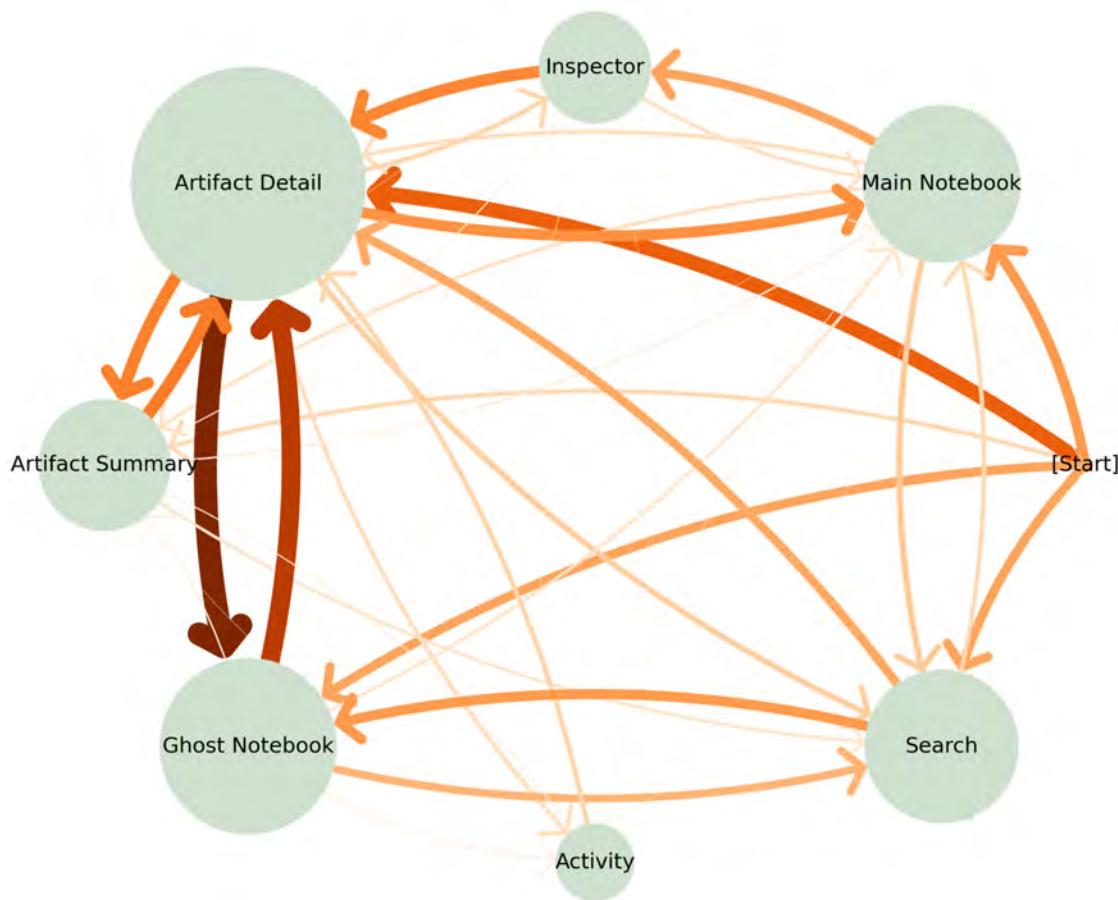
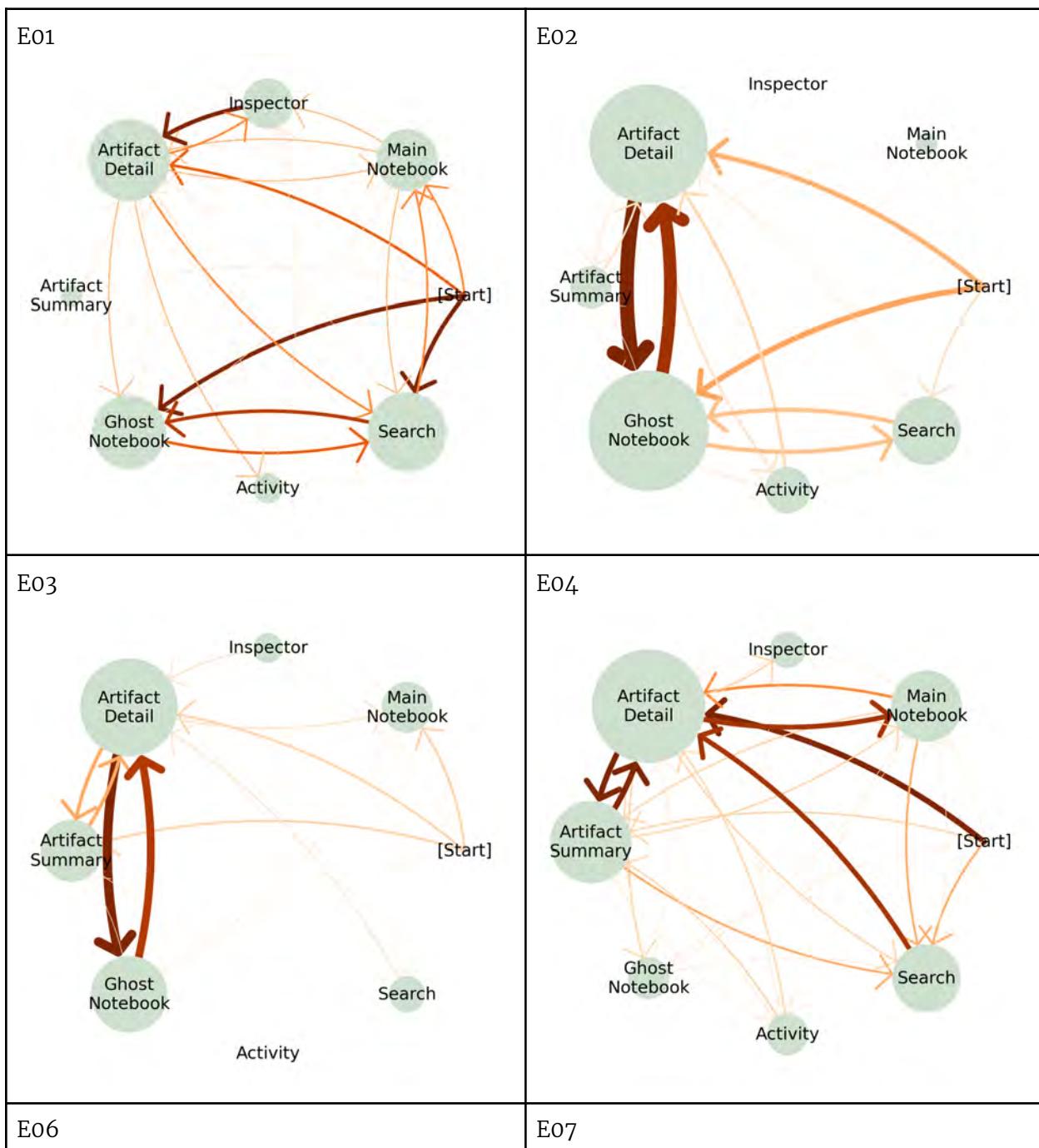
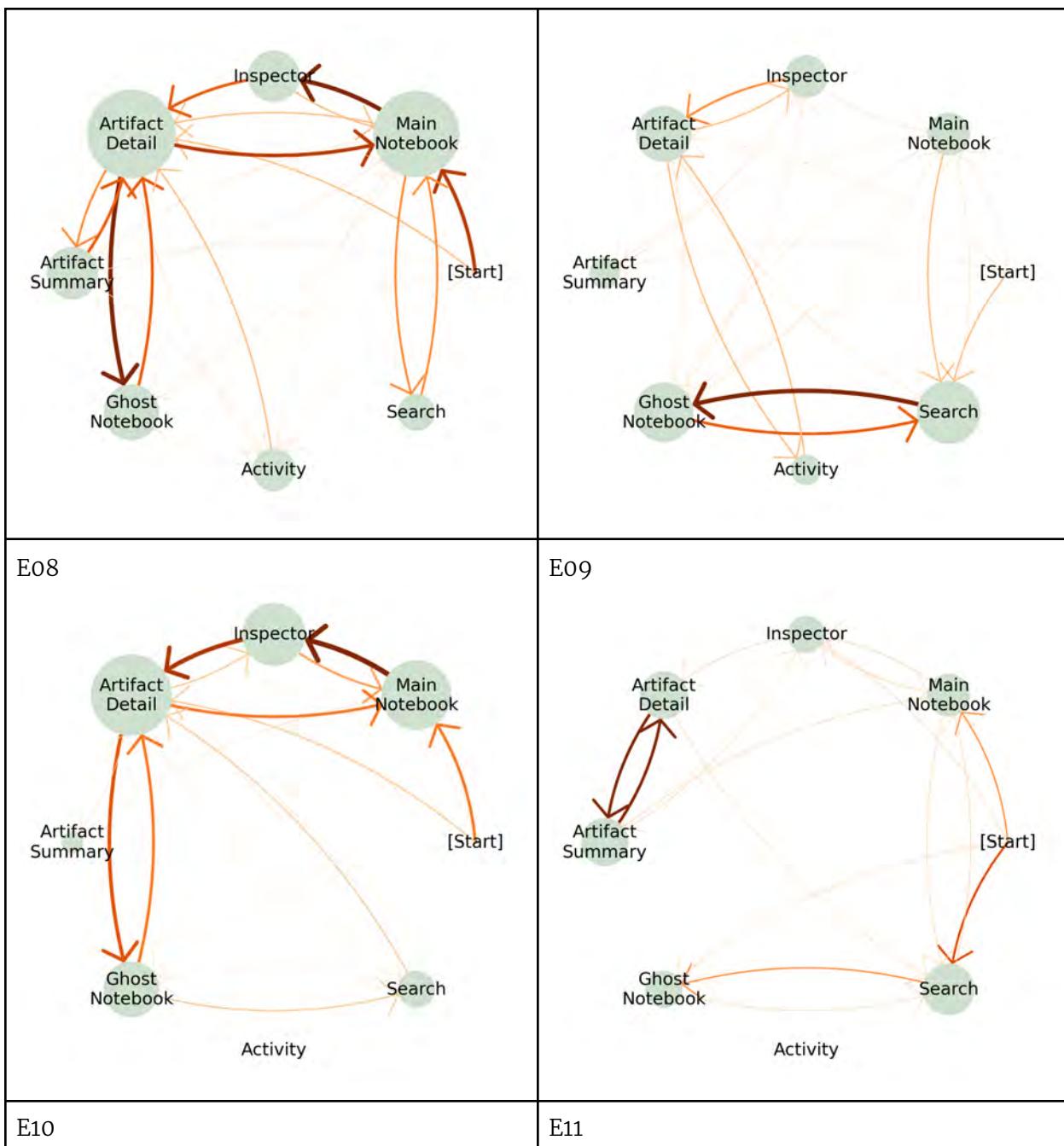


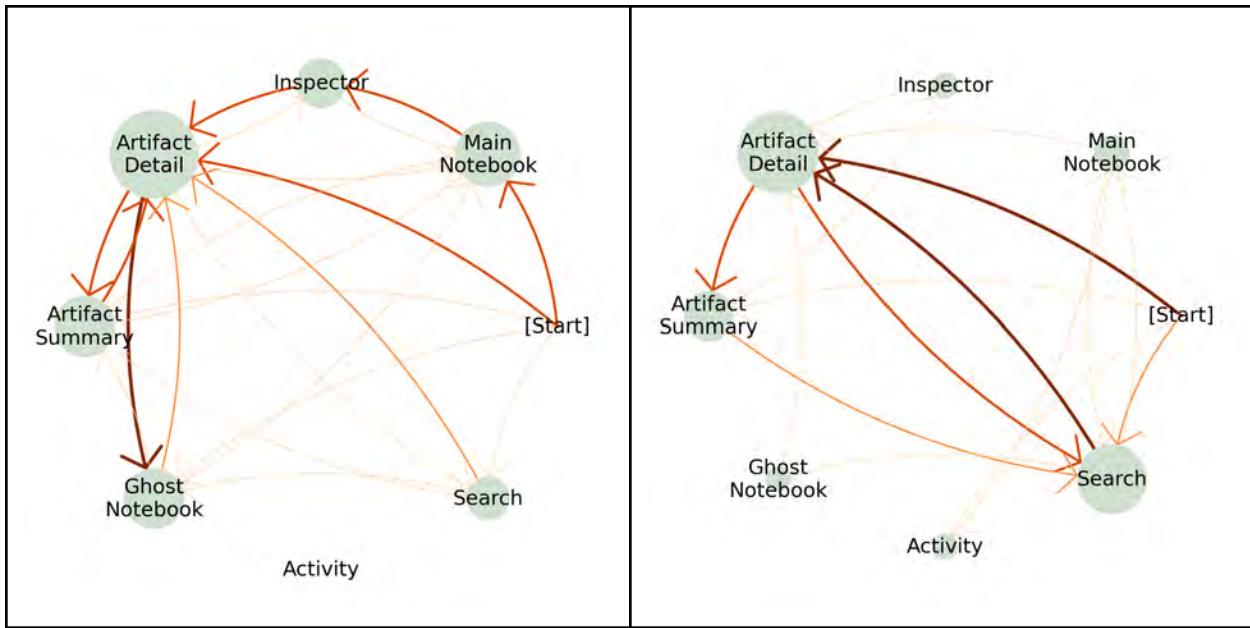
Figure 14.4 Nodes in this graph represent a feature of Verdant (with the exception of “Start” which is the user’s starting point for each question). Node size indicates the total number of visits to that feature, and edge size & darkness indicate how many times participants took that path.

Individual paths are shown below in Table 14.4, which also demonstrates how participants tended to retrace their path through the same set of features to answer multiple questions.

Table 14.4 Feature Pathways For Each Participant. Breakdown of Figure 14.4







The boldest edges indicate several notable pathways:

1. **Artifact Detail** and **Ghost Notebook** are frequently used together. In this usage we observe that participants find an artifact version of interest in the Artifact Detail view, and then open up a Ghost Notebook side-by-side to see that particular artifact version in context of the full notebook as it existed at that time.
2. A cycle between **Artifact Detail**, **Main Notebook**, and **Inspector** indicate a common usage we observed. If a user is in Artifact Detail looking at a particular artifact's version that is *not* the artifact they're looking for, the user returns to their main notebook to find where in the current notebook that history might be. Next, they use the Inspector to click on an artifact in the current notebook, taking them back to the Artifact Detail, which now shows all versions of that selected artifact.
3. **Artifact Detail** and **Artifact Summary** are frequently used together. This is to be expected, since they both share the same UI container of the Artifact tab and contain navigation links between each other.
4. Participants most often start their search at **Artifact Detail**, **Main Notebook**, **Search**, or **Ghost Notebook**. Note that for the first 3 questions of each scavenger hunt participants were requested to start their search at a specific assigned feature: **Artifact Detail**, **Search**, or **Inspector**. Despite that priming, in reality participants started in other places and almost never started at the Inspector. The Inspector situation can be easily explained: in order to “inspect” an artifact with the Inspector, participants first searched in their main Jupyter notebooks to pick out the artifact they wanted to inspect. Sometimes, after browsing the notebook, participants became distracted and moved on to using different Verdant features to answer the question.

What did participants think about Verdant's features?

Following the scavenger hunt, we conducted an open-ended interview. We asked participants for feedback about Verdant: how helpful the tool was, which parts of the tool they liked and didn't like, or any parts they found confusing. Feedback is summarized in Table 14.5.

Although usage logs (Figure 14.3 & 14.4) indicate that the Ghost Notebook and Artifact Detail features were the features participants used the most, only 5 participants [E01, E03, E04, E08, E11] explicitly discussed these in their feedback. Interestingly, participants were enthusiastic about Search and Inspector features.

Table 14.5 Overview of Feedback For Specific Features

Verdant Feature	Feedback	Participants
Ghost Notebook	Very useful for seeing outputs and surrounding notebook	E01, E04, E08
Artifact Detail	The feature to see code and output pairs side-by-side is really helpful	E08
Artifact Detail	Seeing the full history of output is super helpful	E03
Artifact Detail	Really convenient	E11
Search	Very useful, especially for questions with specific keywords or code snippets involved	E01, E02, E03, E04, E06, E07, E10
Search	Overwhelming, returns too many irrelevant results, could use better relevance sort	E08, E09
Inspector	Very useful for quick history access	E01, E03, E06, E07, E08, E09, E10
Activity View	The grouping of activity made it very readable	E02
Activity View	I liked looking at it and having it there	E02, E03, E04, E08, E11

7/10 participants highlighted the Search feature as being particularly useful [E01, E02, E03, E04, E06, E07, E10]. E01 found search best when they could imagine what “code for that snippet would probably look like.” However, the Search had some pitfalls due to its basic search algorithm. E08 found the search overwhelming in returning too many results. This is because one of E08’s scavenger hunt tasks involved finding how a variable “x” had been used. When they searched for “x”, Verdant’s search returned every occurrence of the character x in all text in all the history of the notebook — a massive number of results, very few having to do with the variable

“x”. This is a usability flaw in the current search implementation, which is overall a very basic textual search. E09 also suggested that it would be helpful to order results by relevance. It would also be easy to add word or variable search, as in most real tools.

6/10 participants highlighted the Inspector feature as being particularly useful [E01, E03, E06, E07, E08, E10]. As E03 put it: “*if I have the cell, click on it and it'll tell me exactly what the history was for that thing*”.

Although participants didn’t use the activity tab during the scavenger hunt, four participants explicitly brought it up as a feature they would want to use in the future:

“*On a longer-term project that would actually be really meaningful to be able to say like, Oh yeah. What was it that I did last week? Or how does this look at the end of Friday?*” - E09

How was the usability of Verdant?

Two participants called Verdant “pretty straightforward” and : “*If I want something I can quickly get to it*” [E03]. Our scavenger hunt results do support that participants were reasonably efficient and successful using Verdant to find history facts. There was an overall consensus, however, that Verdant has a lot of features and content to look at, with something of a learning curve [E01, E02 E04, E10, E11]:

“*I do see in retrospect, the usefulness of all the different features, but definitely on first use, I think you ignore a lot of them.*” - E04

Although in S1 every participant completed an interactive tutorial that had them use all the major features of Verdant – and were given this same tutorial to reference in S2 – participants tended to forget about features. Two participants [E01, E04] re-discovered the Ghost Notebook feature only after having done some tasks without it. The Inspector tool was a popular feature among participants, as discussed above, but we had to add to our protocol to verify that participants knew what the Inspector was before the scavenger hunt, as described in Chapter 13. The Artifact Detail view and Ghost Notebook both had more advanced functionality that most participants never encountered. We are less concerned that there is a learning curve for the details like the side-by-side option, and more concerned with making key features like the Ghost Notebook and Inspector easier to discover.

Our logs and interviews together highlighted three additional areas for design improvements: 1) Navigation, 2) Naming conventions, and 3) Horizontal screen space.

Usability: Navigation

Navigation was the core usability problem in our Jupytercon Scavenger Hunt Study (Chapter 11), and the topic of two separate design iterations to make navigation easier (Chapter 12). My impression from observing both studies was that navigation was far smoother in the current

study, with less observed frustration than what we saw in the Jupytercon Scavenger Hunt Study. However, navigation was still not perfect. While 3/10 participants reported that navigation was easy, another 4/10 reported navigation was “*a little bit hard*” in that they tended to accidentally switch screens without understanding how they got there or how to get back:

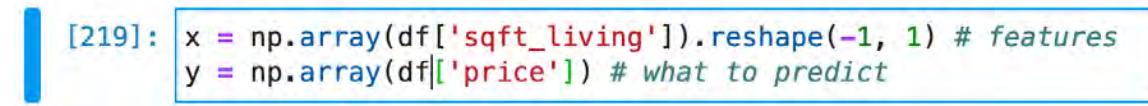
“A couple of times I seem to accidentally move from artifacts to activity, just now, and I couldn't possibly tell you how I did that.” – E02

“I will see the link... but I don't know, I click what's fully going to happen all the time and whether it's something that's gonna pop up or not or when it's supposed to link me somewhere or not.” – E10

“There's not a direct back button, is there? I think that would have been useful for me sometimes. Cause I like would want to go exactly back to the page that I was on, but I couldn't.” – E04

Usability: Naming Conventions

Naming of artifacts and versions in Verdant is again, a design issue we have been grappling with since the Jupytercon Scavenger Hunt Study. The key problem is that each cell in a Jupyter Notebook has a corresponding number in brackets next to it, called the *kernel number*:



```
[219]: x = np.array(df['sqft_living']).reshape(-1, 1) # features
y = np.array(df['price']) # what to predict
```

Figure 14.5 The troublesome kernel number, here 219 in brackets

This number [219] does *not* mean that this code cell is “code cell 219”. This cell is actually the 10th cell in order from top to bottom in the notebook, and the 13th code cell the user created chronologically. Rather, the kernel number [219] indicates that the *last time* this cell was run, it was the 219th run executed on the kernel (the Python 3 runtime) —which may not actually be the *current* kernel. Every time the user starts a new session or restarts the kernel, the execution count resets, but the kernel number next to each cell *does not change* until that cell is run again. Essentially, this number is useless as an identifier: constantly changing, *not* unique, and frequently outdated. Unfortunately, this is the only number users see next to their cell, and many users don't know what it means. Repeatedly during the study, participants mistook the kernel number for meaning “code cell 219”.

Since cells are anonymous in Jupyter notebooks and do not expose a unique identifier or name to the user, we originally followed this design and avoided explicitly naming cells or notebooks for the user in Verdant-1 and Verdant-2. One feedback from the Jupytercon Scavenger Hunt Study was that this was too confusing: users wanted names to refer to, search for, and recall so that they could more navigate history. In Verdant-3 and Verdant-4 we showed the user

something fairly close to how our history identifies artifacts and versions internally. Figure 14.6 below shows how Verdant's naming scheme corresponds to the notebook.

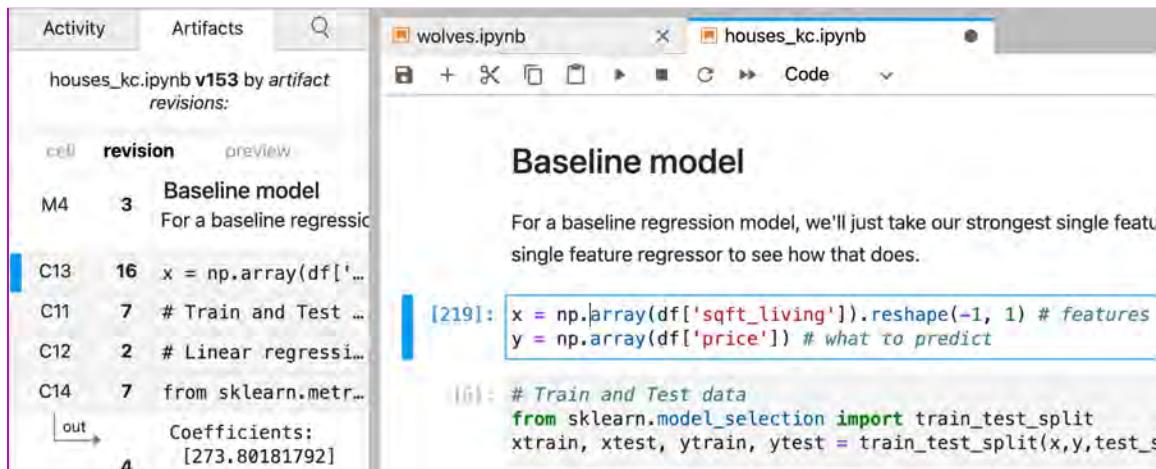


Figure 14.6 Naming in the Artifact Table view. C13 corresponds to the user selected cell in the notebook, and has 16 versions.

The trouble that users encountered in this study is that there is no correspondence between the naming used in Verdant and the kernel number in the Jupyter notebook:

"I was never quite clear about this, this number versus this number. And I kind of just wished that they matched, even though I know that I haven't made 27 updates to block C 11, or I haven't made 139 updates, I've made 27 or whatever. I kind of just wished that it was like, this is C 11 in version 159, rather than two numbers." – E02

6/10 participants reported being confused by the naming conventions. Another source of confusion was the different labeling schemes between artifacts:

"The naming is a little confusing to me. Um, like I said, kind of as we were going through the, like, "C", I guess I just thought that meant cell or something? I don't know what this "R" is, does it mean run?" – E08

From these issues, our current design recommendation is to simplify the naming scheme used in Verdant and then add back in the idea of ambient version indicators in the cell margin (Verdant-1, Verdant-2 Sketch 11.5). Although we cannot change the kernel number design of Jupyter Notebooks, we could add our own number indicator to the margin of each cell that matches the history naming. This number indicator may also give us a solution for improving the version inspector interaction: instead of activating a version inspector button *then* clicking a cell (Figure 12.8), the user could instead click on the history number besides each cell to inspect its full history.

The screenshot shows the Verdant interface with the 'Artifacts' tab selected. A search bar is at the top right. Below it, a header reads 'NOTEBOOK > CODE CELL 13'. Three code cells are listed vertically:

- C13.r16 created in Notebook v127 1:25pm Jan 29, 2021**
x = np.array(df['grade']
y = np.array(df['price']) # wha
- C13.r15 created in Notebook v125 1:25pm Jan 29, 2021**
x = np.array(df['sqft_living'])
y = np.array(df['price']) # wha
- C13.r14 created in Notebook v124 1:25pm Jan 29, 2021**
x = nn.array(df['bathrooms'])

The code snippets are partially redacted with red boxes.

Figure 14.7 Cell naming conventions in Artifact Detail view. Here C13.r16 means the 16th version (r for “revision”) of the 13th code cell that the user created. In practice this was confusing to users.

Finally, two participants [E02, E10] requested ways to override the naming scheme and give custom names for notebook versions and cells important to them.

Usability: Horizontal screen space

Again, horizontal screen space has been a challenge since the very first iterations of Verdant, simply because code/output/markdown content takes up a lot of space. For instance in Figure 14.7 above, note that the Artifact Detail pane only fits part of the overall code snippet width. Due to the scroll bar design on many contemporary browsers and operating systems, it is not visually obvious that the user can actually horizontally scroll Verdant’s pane. As a workaround, the user can expand the Verdant side panel horizontally, which we observed 8 participants did a total of 20 times during the scavenger hunt.

What Went Wrong When Users Struggled?

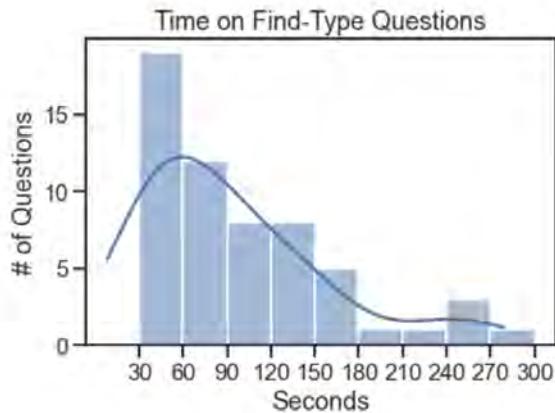


Figure 14.8 Distribution of Time Participants Took to Answer Scavenger Hunt Questions

In the distribution of task times to find the correct point in the history, shown above, we observe a bump around the 4 minute mark. We hypothesize that this bump, and the long tail of task times, indicates possible **failure cases with Verdant**. We took the 8 questions that ran longer than 1 SD away from the mean (> 156 seconds), and did a further detailed analysis of what may have gone wrong in those questions.

In 3/8 slow questions, video and log data shows software bugs in Verdant as the culprit. Throughout the study we fixed software bugs in Verdant as quickly after they were observed as possible, but in these cases, features in Verdant not responding correctly slowed participants down.

In 1 slow question that ran over 4:29, a bad wording in the question sent the participant looking for the wrong thing. In the question “*Find: Go back to the plot where you had $x = usd$ goal bin, $y = project count$, and $hue = main category$* ” the actual variable name for y was **Num_samples**, not “project count”, although **Num_samples** did indeed *semantically* represent the count of projects. While this was our own study design error, it does hint that history search may be brittle to a participant fixating on specific keywords that represent a concept.

In 1 other slow question, the concept a participant was looking for had too many possible keywords or locations. The question was: “*Which variable combinations did you try looking at the relationships between?*” After 4:01, the participant concluded that they had probably identified “most” of the variable combinations they had analyzed, but they had no way of knowing for sure. We believe the question is the culprit. To find all variable combination analyses would require exhaustively searching each variable and browsing through all results – a situation very similar to the “brute force” search scenario we encountered in the Jupytercon Scavenger Hunt Study (Chapter 11).

2/8 slow questions appear to stem from some bigger usability issues already discussed above. In 1 question, the horizontal width of the screen was too narrow such that the user couldn't see the diff change highlighting in an artifact's code, leading them to initially miss the change they were looking for. Instead, they continued scrolling up and down through the artifact's versions for much longer than needed before they horizontally scrolled to see the diff marking the correct version. As previously discussed, in 1 question E04 gave up on answering the question all together, because the history of a particular variable was spread across multiple cell histories, since that code had been moved around.

Finally, the last slow question was caused by a gap in Verdant's history model, that we frankly had not expected to see in this study. Verdant does not capture any kind of runtime information currently, or do any kind of dynamic analysis to trace the exact runtime provenance of results in the notebook. The question E03 attempted to answer is: "*Find: What was the best accuracy score you got for logistic regression?*". E03 quickly found the spot in history that contained their accuracy scores. However, the accuracy scores were listed for a variable called `model`, and there were two different cells containing two entirely different models, *both* using the same variable name `model`. Which model had actually been run to produce the accuracy score? Participant E03 took an educated guess, but without knowing the actual value of the variable `model` at the time accuracy was calculated, there is no way to know for sure in Verdant. To fix this issue would require Verdant doing additional dynamic analysis to record runtime provenance. There are multiple existing tools that demonstrate solutions for this problem in notebooks, e.g., [Head et al. 2019] or [Macke et al. 2020].

PARTICIPANT BELIEFS & VALUES OVER AUTOMATIC EXPERIMENT HISTORY

One value of versioning *automatically* on behalf of a data scientist is that it provides a safety net such that a user never loses their work. 8/10 participants mentioned *recovering deleted content* as an important benefit of Verdant. It is normal in the process of experimentation that people drop unsuccessful ideas as they work towards a solution. In our first study of notebooks (Chapter 4) we observed that notebook authors prefer to tidy notebooks as they work, keeping only the content that tells a clear story [Rule et al. 2018, Head et al. 2019]. A minority of authors try to keep a record of everything they try in their notebooks (Chapter 4). Here 3/10 participants said they normally write exploratory notebooks with the "*keep everything*" [E02] mentality, but despite that self-report we observed that those participants did not actually keep *everything*. A crucial distinction is that, whether through notebook structure or Git or other history-keeping methods, data scientists keep *what they think is valuable at the time* (Chapter 3 & Chapter 4). Dead-ends, disproven hypotheses, failed experiments, variable relationships with no correlation, etc., are reliably and routinely deleted.

Should we care about deleted content? Our observations suggest that deleted dead-ends and discarded hypotheses can be important later, even if only to let a data scientist explain why they *didn't* go with a certain approach as discussed next.

Data scientists cannot always anticipate what will be useful

As reported in prior studies [Yoon & Myers 2012], programmers are not perfect at anticipating what they might need later, especially in exploratory situations where ideas are rapidly evolving:

"Um, and that's something that particularly bothers me when I go back to a notebook a couple of months later, because I'll be like, Oh, I think I have this thing, but then go back to the notebook and realize that you've erased it, replaced it with something else.... Or if I remember, I'll try to save the graphs into a separate folder so that when I want to look back at it, at least I have it. But honestly, whenever I remember to do that, chances are that I don't need the previous history. It's only cases when I don't remember to save the plots that I need the history." – E03

When something is lost, the solution is usually to recreate it from scratch:

"And then of course I ended up later, like two weeks later being like, "Oh, but actually I need to look at that question again" and then I have to rewrite it." – E01

The need to recreate lost work was also the most reported pain-point due to lack of history in our Query Design Survey from Chapter 4.

Why was that a dead-end again? Recalling details is hard without history

Per our study design, we spaced out S1 and S2 of the study by 10–14 days with the aim for participants to forget just enough about their S1 work that they wouldn't be able to answer all of the history questions just from memory. Besides analysis questions about what the participant did during S1, we also mixed in a question about something the participant *didn't* explore during S1. As reported above, in all but 3 of 64 scavenger hunt questions, data scientists used history data to answer questions rather than answering from memory alone. We did not observe any differences between participants answering questions about something they did do versus something they did not do. When asked about something they did *not* do, e.g., *"Did you use or look at the country feature at any point?"* participants often expressed uncertainty such as *"I don't think I actually did... but maybe"* [E10] before searching in Verdant for the answer. However, importantly, participants would respond in the same way to something they *had done* in S1, that they had deleted and then forgotten about.

We heard from 4 participants, who are all professional data scientists rather than students, that in real data science practice they do get plenty of questions from team members or clients about possibilities *not included* in the final cut of their analysis. Questions such as “What’s the

relationship between <T> and <Q>?" or "Have you tried <such and such a plot>?" or "Why did you not include <M> as a feature in the model?" or "What was the performance like with <this model>?". If there was no interesting relationship between variables <T> and <Q> in the data, it's possible that the data scientist deleted that investigation.

"I can see [Verdant] being super helpful too for describing process. Or, you know, six months later when a reviewer wants a revise-and-resubmit, and you're like, what did I do? You know, it's really common for a reviewer to have a question of "did you try this?" And, you know, the answer is sometimes yes, so I can definitely see the merit of [Verdant] there." – E02

History is a collaboration tool

We asked participants to describe more about the social contexts in which they would or wouldn't use history.

One unexpected idea we heard from participants was that a data scientist gets more use out of examining history in team settings than if they are working alone:

"So essentially the more people involved that are questioning your decisions or your questioning theirs, I think that's where it's the most useful. Because if I am doing it on my own, then I may not even be aware of what I'm missing, right? Like some questions I may not even ask, but it's when, whenever more people are involved in the project, that's where I think it would be the most useful. And, especially if newcomers, you know, come and look at it and don't have much of a context and it's a little bit, "why didn't he do this or that?" And if you want to say, 'Oh, maybe I did try it. Let me check back,' then that would be useful to like onboard newcomers." – E10

Participants who worked in settings where other people besides them were involved in data analysis reported getting history questions in their daily work, whereas participants who did solo data work less so. The key users for history were collaborators or newcomers to a project who needed onboarding. In contrast, E01 reported that they would not use history, even to answer history questions, in front of clients due to professionalism:

"I don't think I'd pull up the history in front of a client. Um, because honestly for a client, I think the better thing to do there would be to answer to the best of your ability just verbally or with the stuff you already have up. Then you can always send a followup afterward that's polished. I wouldn't want to get into something that you got to fumble around a bit. For internal project meetings that might be doable, to bring up, especially like past plots or something. It really honestly depends on who the people I'm presenting to are and what level of polish I need to have." – E01

In terms of onboarding, E08 talked about wanting to use history to understand what their teammates had done in their work:

“Everyone kind of structures their analysis in slightly different ways so that, you know, figuring out what they did is really hard. I spent a little bit of time emphasizing this point of like, I go do a bunch of things, but then only some of it makes it to the final cut. Being the person who does that, I know what I've excluded. Um, but if I'm picking up something that someone else has done, I have no idea what else they've looked at.” – E08

However, E08 also cautioned that since understanding the final-cut of someone else's analysis is hard enough, exploring their history might just be too challenging.

These quotes are encouraging clues that collaborative history for data science history, and summarizing one person's history to be examined by another, are interesting areas for future work [Zhang et al. 2020, Wang et al. 2020].

Most important history? Model metrics & Plots

Many of our participants had history keeping habits in their data science practice. 8/10 participants used Git to store their notebooks and another 5/10 participants talked about saving images, plots, or model metrics to external folders or files. For code, community best-practice dictates a clear solution: store code history in Git. However, for other things like plot images, notes, or model metrics, participants stored those in a various assortment of separate places. For a single project model metrics might be stored in a csv (E10), code in GitHub, plot images on Google Drive or Dropbox (E02, E10), then loose notes and results jotted down in a Word document (E04). The difficulty with history and artifacts spread to so many places is that they're harder to tie back together. The benefit of a cohesive relational history like Verdant is that context is preserved. 6/10 participants explicitly talked about Verdant's style of history being beneficial for preserving plots with a clear way to reproduce them. 5/10 participants explicitly talked about Verdant's style of history being beneficial for preserving a history of model metrics. Once modeling reaches a more mature phase of iteration, best-practice suggests model metrics be automatically logged, such as E10's reported normal workflow where each model's settings and metrics are logged to a csv file. However at the earlier exploratory phase, where less formality is warranted, Verdant's style of automatic history gives users quick access to flip through model metrics in the history of a cell output.

Careful, don't collect junk!

Although our results suggest that automatic relational versioning such as Verdant provides substantial benefits to data scientists, there are tradeoffs. The drawbacks of versioning automatically are 1) storage costs and 2) potentially more versions than are helpful. These issues and details of Verdant's versioning implementation are discussed in Chapter 9. Two participants in our study were overall hesitant about the idea of automatic versioning, both worrying that it would store many more versions than necessary. Both participants thought that Verdant should *not* store history in cases of syntax errors or debugging, which Verdant actually

does use some heuristics to avoid storing. For these data scientists, they preferred a model closer to Git, where a user could manually check in more semantically meaningful versioning of their notebook. It's worth noting that Verdant's user interface and interactivity would work the same for a manual version without automatic history. This is a possibility worth exploring in the future. Finally, one participant with these reservations did later shift more favorably after working with Verdant:

"I really like it. I think it is helpful. Um, and it's, it's one of those tools that as a user, it seemed very cheap. It seemed like it wasn't really slowing down my computer, it wasn't making anything worse or slower or difficult. It wasn't hindering me in any way. And the potential benefit is pretty big. Um, if I did want to go back and check that out. So it's definitely something that I would kind of just happily leave running, for the day that I did need it. Yeah, I think that'd be a really nice safety net to have." – E02

CHAPTER CONCLUSIONS

For us as researchers, this study served to show that Verdant is *effective* at what it is designed to do, and also served to crystalize what exactly the value of experiment history is to practitioners. All along we have been designing Verdant and its prior history tools with the hypothesis that a more complete and easy-to-use history support would be helpful to practitioners. We saw in these results that practitioners were able to answer real history questions successfully with Verdant in 98% of cases in an average of 1:26. This is highly encouraging evidence that Verdant is a practical design for data workers to ask questions about their experiments in short enough time to fit appropriately within a meeting or programming session. Moreover, in qualitative results we heard what a complete history enables practitioners to do. History means that plots aren't lost. History means that a data worker can more easily onboard a colleague to their project. History means that a data worker can just as easily show evidence for why an experiment failed as for why an experiment succeeded. With no upfront effort from the user to collect any history, Verdant provides an effortless safety net to ensure that experimentation people do is preserved and can be easily interrogated later. Like all our prior studies, this current study revealed areas for improvement and further design work. However, we consider these results to be validation that our design research can serve as a solid foundation for future work in experiment history.

Part IV: Future Work & Conclusions

Chapter 15: Future Work

INTRODUCTION

So where does this leave us? The work in this dissertation has been iterating towards a vision where a data scientist e. ortlessly has the history of their experimentation at a moment's notice, and can effectively use their history to explain their prior work. We believe that our most recent results in the Verdant Usage Study suggest we've come close to achieving this. Practitioners in our study were able to locate an answer for 98% of the history questions we posed in an average of 1 minute 30 seconds per question, using a fully automatic history tool. While certainly we believe that these results can be improved with further iteration, to us this demonstrates the value and viability of interactive and automatic versioning for experiments.

The Future Work will be split to address two major avenues: 1) what next steps does Verdant-style history tooling need to be maximally viable in the real world? and 2) what research avenues does Verdant-style history tooling open up? We believe that the most exciting open research questions start once we can assume that data scientists have a complete history. If every practitioner and team has a wealth of data about their iteration and practice, this data may be used in the future for higher-level automation support.

Verdant in the Real World: Design Considerations for In-Situ Interactive Fine-Grained Versioning

Verdant may be able to directly benefit practitioners today, which is why we released Verdant as a JupyterLab extension. However, realistically Verdant and Verdant's style of interactive history support will need improvements in order to support wider adoption, as discussed next.

Storage & Efficiency

First, fine-grained in-situ versioning, like Verdant provides, needs to be more efficient. In this work we engineered Verdant such that it was performant enough to work in real time without slowing down the user's environment on a high-end personal computer. However, Verdant is certainly not as efficient as it would need to be to support long term usage and large outputs. During runtime, Verdant needs to quickly access any version of any artifact to display summative visualizations and quickly respond to user requests to examine the history of a certain artifact. Verdant also needs to support search over the history. Meanwhile, Verdant is recording new history of text and mixed-media output in realtime as the user edits and runs their notebook. Our approaches for storage are discussed in detail in Chapter 9, and would need revisiting to make Verdant performant in real usage. Similarly, as discussed in Chapter 9, we originally included versioning at the granularity of code snippets instead of just code cells, but ended up removing this feature due to performance. Feedback from participants in our Verdant Usage Study suggests that history of code snippets is actually quite important for many users.

Often users move code *between* cells or by combining cells, in which case Verdant currently does not track the provenance of those code snippets and their history is lost. Thus, further research is needed to find an efficient way of incorporating back in some level of code snippet versioning to meet this need.

Git Integration

We believe that the key interaction concepts in Verdant could be readily adapted to work in other coding environments and with other history back-ends — including integration with Git. The ability to interactively look at history in-situ in an active code editor is a familiar concept with contemporary tools like Git and Visual Studio Code. Where Verdant makes a substantial improvement is that we design new interaction forms specifically tailored to *using* data experimentation history. The key integration hurdle for layering Verdant’s style of interactive visualizations on top of a Git back-end is that Verdant breaks down a user’s file into sub-component artifacts, tracked over time. There are a number of ways this could be adapted to work with Git, with the simplest likely being to let Verdant keep its own supplemental history file. Verdant currently has one single big history file `.ipyhistory` where all history is kept. It would be storage-inefficient to commit the `.ipyhistory` file as a normal Git user’s file, since that would copy Verdant’s database again and again across commits. Instead, it may scale better for Verdant to keep a single `.ipyhistory` file per *Git commit*, that just contains the fine-grained Verdant history of that specific commit.

Manual vs. Automatic History

One question that comes up almost every time we have presented this research to a research audience or our study participants is: do we really need an automatic history of *everything*? Clearly, automatically collecting all of a data scientist’s experimentation, including code, notes, and output, incurs storage costs. In Chapter 14 we discuss the pros and cons of automatic versioning from a user’s perspective. Ultimately, however, we are of the opinion that the benefits and safeguards which automatic versioning provides to users outweigh the storage costs. That said, there are ways in which better balances between manual and automatic history could better suit users’ individual needs.

First, as in most software tools, configuration is important. Verdant collects output, code, and everything in a user’s notebook because we have consistently found in user feedback across our design iterations that *different practitioners want different things stored*. For some data workers, the ability to automatically capture output, and the code needed to reproduce it, is the biggest value added by Verdant. For other practitioners, they are not interested in output for their particular project, or *cannot* have output captured for legal reasons, such as United States health data laws. So configuration around what is captured by automatic versioning would be beneficial. Configuration around how *often* content is captured, or the ability to turn off and on automatic versioning as needed, may help data workers feel like they are capturing more semantically useful history. All of the visualizations and interactions provided by Verdant will work the same regardless of how much historical data is captured.

Next, users should have the ability to manually annotate the automatic history. Practitioners frequently requested the ability to name or annotate important versions, instead of having to

remember the system-generated name for a version they know will be useful later. Throughout our iterations of Variolite and Verdant 1 and 2, we implemented different forms of favoriting, pinning, naming versions, or letting users leave comments on versions themselves. These features were dropped from later iterations of Verdant (Chapter 12) due to limited engineering resources, because tagging and annotating seemed less critical to our research goals at the time. In our final version of Verdant, we believe the most obvious form of annotating history would be to let users add something like a commit message to be shown as the title of a version. There is a breadth of possibilities for manual annotation, manual pruning, manual organization, and meta-analysis we could let users do to manage their own histories. Future work is needed to understand what forms of annotation and curation would provide the most value to users.

Collaborative Authoring & More Complex Histories

Verdant and all of our prototypes were designed for the use case of a *single individual* writing code, and potentially showing their work to others. We did not address collaboration in authoring, simply because we needed to scope our research for this dissertation. However, in real practice, collaboration is an important part of data work [Zhang et al. 2020]. Collaboration is also an important part of traditional version control systems like Git. Unfortunately, collaboration not only adds social aspects to system design, but also adds much more complexity to history. Where multiple people can be authoring and modifying the same document asynchronously in parallel, just like Git, Verdant will need to model and visualize multiple contributors and multiple branches. Existing visualizations from Git demonstrate clear design patterns that a fine-grained experiment version control like Verdant could readily adapt. It is less clear how the overall usability of Verdant would be impacted by branching multi-author histories, since a complex history structure was *already* a major usability design challenge in Verdant. Users in the Jupytercon Scavenger Hunt Study (Chapter 11) and users in the Verdant Study (Chapter 14) all reported navigation issues. Since Verdant breaks down the history of specific artifacts, it can be challenging for users to interpret how histories of different artifacts relate to each other over time. Navigation, and navigation of more complex histories, will need to be addressed in additional design and visualization research. Recent work by Weinman et al. in “Fork It” tested displaying branching points in the notebook as side-by-side lanes of work in the notebook [Weinman et al. 2021]. However, this design faced similar navigation problems to what we faced in our designs in that horizontal screen space limits users to seeing just 2 alternatives side-by-side in most cases [Weinman et al. 2021].

Relating Histories of Artifacts Outside of Notebooks

A chief motivation for us scoping our research in this dissertation to computational notebooks is the barrier of relating histories of artifacts coming *from different tool sources* in a data worker’s environment (see our failed Rose Quartz prototype in Chapter 8 for discussion). Even though Verdant captures *many* of the artifacts that contribute to a user’s experiment within a notebook’s environment, we do not even attempt to capture the history of the user’s *data*. Our reason for *not* dealing with the versioning of datasets, is that data versioning requires its own set of specialized tools and approaches, e.g., DVC¹. With Verdant we chose not to coordinate with other tooling outside of a notebook’s internal environment. Coordinating history between

¹ DVC Data Version Control <https://dvc.org/>

different tooling, different media-specific repositories, and recording workflow relationships between disparate histories is a major barrier for future work.

Beyond Verdant: Using History Data for Higher-Level Forms of Support

The practitioner use cases that Verdant serves stay quite close to the history data itself: how to help practitioners effortlessly store experiment history, and how to let users effectively retrieve what they need out of their histories. There's much more that can be done beyond storage and retrieval, *once* we have a base system like Verdant collecting rich datasets of experiment history data. This is somewhat another argument in favor of automatic history collection over cheaper manual history commits: more data collected allows us to do more powerful analysis and modeling *on that history data*. Next, we will discuss several promising avenues that use analysis on top of history data to provide higher-level forms of support.

Detecting & Communicating Story and Rationale out of History Logs

The logs and visualizations we provide in Verdant are low-level in that they are at the level of the exact edit logs we record from the user. Although we use interaction and visualization techniques to help users extract semantic history facts from these logs like “What happened when X feature was tried?”, this is not the same as *summarizing* history. Also, as we discuss in Chapter 9, not every version of a programmer's code can truly be described as “an experiment”. Some very typical programming work, like debugging or fiddling with syntax, do not actually contribute to the narrative of how an analysis was done, but nonetheless clutter Verdant's logs. While we apply some filtering and heuristics to improve log quality (Chapter 9), the fact is that Verdant's logs record *nearly verbatim everything* the user does.

Another motivation is that in our Verdant Study, multiple practitioners were excited about using Verdant's style of history to onboard newcomers to a project, or to attempt to understand someone else's work. Recent work on collaboration in data science [Piorkowski et al. 2021, Zhang et al. 2020] has also stressed the need for better and easier forms of communication about the rationale behind data work. When the rationale behind an analysis or model decision is held only by the original author of that work, information is easily lost when data workers move companies or want to reuse analyses outside of their original context. Zhang et al. found that among professional data workers at IBM, 84.6% of notebook users expected that their notebook code would be reused by someone else [Zhang et al. 2020]. History data contains much of the information needed to understand the process behind data work, in a permanent format. How might we support *summative* understanding of a chunk of history?

Recent work has made progress in using machine learning to summarize data analysis work. For instance Wang et al. used modeling to automatically suggest markdown documentation to data workers in notebooks [Wang et al. 2021]. Meanwhile Ge Zhang et al. showed promising results in being able to automatically label the category of data science activity going on in a notebook code cell [Zhange et al. 2020]. We are optimistic that as these techniques advance, Verdant's style of history logs might be better clustered, categorized, and summarized to provide users with a high-level overview of what went on in the development of data work.

Higher-level History Question Asking & Answering

In addition to history summarization, this same kind of automatic detection of semantic information from activity may also support better history search. Currently in Verdant, our interactions and visualizations serve to help users find the history of particular artifacts based on memory cues like time, location, keywords, or visual features they might recall (see Chapter 11 for discussion). However, it is really up to the user to operationalize their actual question into specific versions and artifacts that let them answer that question.

Recalling our Query Design Exercise from Chapter 4, we find that Verdant reasonably serves just over half 56% of all history tasks practitioners articulated (Appendix G). Why only 56%? Verdant works well for finding specific things. Many of the remaining questions Verdant does not cover are too high-level to answer with a specific version of a specific artifact. For instance: “*List of all significant predictors of [insert name of variable Y]*” requires a semantic understanding of which features out of all the analyses an author tried were “predictive”. Now it may be for some of these high-level questions that the practitioner might be able to articulate the same question into a finding task that Verdant can achieve. However, from the raw text of participants’ history questions alone, there are 35% of them that require some understanding Verdant does not have.

With better machine semantic understanding of history, future work may be able to answer higher-level questions for the user directly. For instance, a user might ask “Why did we choose this parameter?” by pointing at the spot in the code, similar to the interaction in Whyline [Ko & Myers 2004]. Currently Verdant could deliver the history of that particular spot of the code, but would not deliver other relevant information needed to answer the user’s question if it were not contained in the history of that specific artifact. Better semantic processing, including static and dynamic data flow analyses and built-in knowledge about what methods do may be used to help users understand the entire scope of artifacts and versions needed to answer their questions.

Highlighting gaps in the user’s exploration

One of our earliest ambitions in this research was: can we use the practitioner’s history of their work so far to suggest what to try next or otherwise make the experiment process more robust? With a poverty of available history data, this avenue was not viable to us at the time, but an actualized tool like Verdant makes it possible to collect sufficient data. Recently in the research community there has been a lot of interest about data science experimentation, analysis or model building workflows [Lee et al. 2020, Sanchez et al. 2021]. In our own studies, we have observed that the data science exploratory process is nothing terribly precise: at any point in time a data scientist will try the most promising avenue at the time, in a kind of greedy-algorithm style walk through of an immense decision space. In our Verdant Study, we saw that when we gave 11 data scientists the same data and the same task prompt, they varied in the approaches they took and number of insights they found. Liu et al. similarly found in a study of published research analyses, that experimentation decisions tended to be very personally driven by the researcher doing the analysis [Liu et al. 2020]. They proposed helping researchers visualize the *multiverse* of their analyses: essentially show in a visualization what path of

decisions they took through a much bigger decision space of approaches they *could have taken* [Liu et al. 2020].

On the one hand, a human data scientist’s domain knowledge and common-sense reasoning guides their exploration more effectively than a machine alone. On the other hand, techniques like parameter optimization and AutoML approaches can do much more experimentation covering more of the decision space than a human alone [Xin et al. 2021]. Many researchers estimate that the future of data work and machine learning will be a collaborative process of programming between the human and automated methods (like AutoML) to reach the best results [Wang et al. 2019, Wang et al. 2021, Xin et al. 2021]. Can we make the process of experimentation with data more rigorous by bootstrapping human data scientists’ efforts with automation support?

History logs of experimentation are a promising ingredient in this envisioned collaborative future, because based on the *history* of what a person has done so far with their data, automated approaches could more effectively assist appropriately with the context of what a person is attempting to do. History logs could also be used in the visualization style of Liu et al. [Liu et al. 2020] to semi-automatically chart the decisions a data scientist has made so far, and highlight possible gaps. Importantly, this usage of history logs rather depends on the research direction presented in the prior section, in being able to extract semantic decisions and motivations out of raw edit logs.

CHAPTER CONCLUSIONS

This chapter outlines some specific avenues for future research related to experiment history, but broadly we expect that the data provided by rich experiment coding history may enable many more new types of visualizations and user interactions that we have not yet anticipated. Zooming out even further to thinking about experiment history data as a form of programming logging and *worker* logging, we expect that fine-grained history data like this will be part of a broader conversation and research agenda into how sensing of worker activity should be handled. Like any activity sensing data, our experiment logs have the potential to be misused to invade data worker privacy or invasively monitor a data worker’s programming practices. While policy is outside of the scope of this dissertation, we want to acknowledge our work as falling under that societal concern, as more and more data is collected about workers.

Finally, we hold hope that Verdant and our body of design work in this dissertation will inspire future systems of this kind in the real world. The design ideas and findings we uncover in this dissertation may apply to history in a broader set of data related work, or even help inform avenues in other forms of experiment-driven work like design or engineering iteration histories.

Chapter 16: Conclusion

In this dissertation we have described a detailed investigation of history support for exploratory programming data work, carefully aligned with practitioners' real world needs. We conducted two interview and survey studies in different contexts to document data workers' current practices around code and data experimentation. We designed and engineered a series of 5 prototype history tools, culminating in Verdant-4, released as an extension for the JupyterLab notebook environment. During the design process we conducted a series of four usability studies to test that our proposed design interventions were usable and closely aligned with practitioner needs. Finally, in an observational study of Verdant's use in exploratory programming practice by data workers, we showed that our history tooling design allows users to collect all of their experiment history without any effort, and successfully find semantic information about their work history in an average of 1 minute 26 seconds.

Our thesis statement at the beginning of this dissertation was:

Data work frequently involves exploratory programming which requires a new kind of versioning for history and new interaction techniques for exploring that history, which can help data workers more effectively answer their questions about what they explored.

This thesis was achieved by developing new history interaction techniques and approaches, which we then validated in usability studies. Although we cannot say that the problem of good history support for exploratory programming data work has been solved by this dissertation, we have demonstrated substantial progress. Of all the queries generated by data practitioners (Chapter 4), our final system Verdant-4 can help practitioners answer 56% of them (Appendix G), which represents all history tasks that do not require history of a dataset and can be solved by finding a particular historical artifact. As shown in our final Verdant Study, (Chapter 14) practitioners can answer questions fairly quickly (1:26 on average) with precise history evidence.

As a whole this research contributes to the interdisciplinary field of Human-Centered Data Work as well as to the field of Human-Computer Interaction (HCI). Our studies of practitioner work practices (Chapters 2,3,4) have been broadly cited across interdisciplinary research communities interested in data science workflows and human-centered tooling for data work. Our findings show that real data work is more messy, experimental, and iterative than had been depicted in prior work:

- Exploration adds risk of investing in an idea that may fail or be discarded. For this reason many practitioners prioritize finding a solution over writing high-quality code.

- Practitioners commonly comment-out code they want to preserve but don't want to run. Keeping alternative code through commenting, duplicating code snippets or functions in the same file, or duplicating files are all forms of “informal versioning” tactics practitioners use.
- Many otherwise active users of version control software like Git choose not to use version control for managing their experimentation and choose to rely on informal versioning tactics instead because they need fast access to *multiple* versions of *parts of* their experiment at once.

Additionally, our Notebook Usage Study is one of the very first published studies documenting how real practitioners use notebooks for active data work, and thus has more broadly informed new research on computational notebooks. Key findings from this study include:

- Besides their usage as a shareable document for communication, notebooks are also used as scratchpads for testing out quick ideas, and testbeds for models that are then transitioned into production.
- Adding, deleting, combining, and organizing notebook cells is a natural part of iteration cycles in authoring notebooks. Cell patterns include “expand-then-reduce” where an author creates many tiny new code cells to piecemeal build up some functionality, and then “reduces” all of those cells into just a few condensed final cells that encapsulate the final result.
- Whether they use markdown to create narrative is a matter of personal style, but most data workers use minimal markdown headings or code comments to document their in-progress data work. This is in contrast to other usages of notebooks, such as for interactive textbooks or tutorials, where heavy narrative markdown positions a notebook as a literate programming document.
- Using cells, notebook users have their own informal versioning tactics for managing their experimentation, just like non-notebook users.

As HCI system contributions, our tools Variolite and Verdant demonstrate novel functionality that substantially improves user experience with experiment history, and may also be useful in other contexts for exploring versions. Where previously practitioners had very limited ways to interact with their experiment history, restricted to viewing older commits or older copies of their work, our designs contribute:

- An automatic history model which passively records all of a user's experimentation in the background while they work, providing data workers a full history record without any manual effort or noticeable computation time overhead.
- A history UI sidebar allows a practitioner to quickly retrieve the history of any specific part of their code, notes, or output, displayed side-by-side with their active computational notebook.

- A stream of “minimaps” concisely shows users at-a-glance what happened in each version of their work, and how their notebook has evolved over time.
- A “Ghost Notebook” provides a full historical view of any notebook version so that a practitioner can see a particular artifact version in its original context, or compare older notebooks to their current active notebook side-by-side.
- A history search bar allows users to run text searches across their history, for when users recall keywords for work they are trying to recover.
- A Version Inspector provides easy point-and-click access to the full history of any cell or output. This is most helpful when practitioners know the location in their notebook that likely contains the history they are trying to recover.

Overall, the research approaches and trajectory of projects undergone in this dissertation also carry some broader takeaways. First, our investigation of history for experimentation engages with practitioners’ “messy” programming practices, poor organization, time-pressure, forgetfulness, and other elements of human fallibility within data workflows. Many data workers we interviewed definitely did not follow all best-practices for managing their experiment history or engineering their analyses and models in a robust way (even when they were familiar with these techniques and practices). As research in areas such as end-user programming (EUP) have argued for a long time, programmers are people too [Ko et al. 2011, Myers et al. 2016]. For data work specifically, we believe that a key element of ensuring our societal values for responsible AI, ethical data work, transparency, etc., are met is ensuring that human data workers are adequately supported. An empathetic HCI lens can help us use the practices that people actually follow to identify important design directions where practitioners actually need more support.

A second theme of this dissertation is the amount of iteration it takes to get from a compelling proof-of-concept research prototype to a design that can actually withstand the stresses of real substantial usage. Consider that our early prototypes Variolite (Chapter 7) and Verdant-1 (Chapter 10) were both positively received by participants in our studies, and the research community who gave awards to both systems’ papers. Both systems were important steps, and contributed design elements we carried on to later iterations, but ultimately both systems were still quite far from achieving our design goal in reality. Subsequent iterations Verdant 2, 3, and 4 required substantially more engineering investment, but also far more close and detailed consideration into how our history interactions would work in practice. Truthfully, the years of work in Verdant 3 and 4 felt oftentimes less like research and more like product development. Ultimately, however, only through the maturation of our designs and engineering were we able to put Verdant into realistic usage with real practitioners to provide evidence that: Yes, usable experiment history is achievable through this design direction. Yes, given a usable form of experiment history, practitioners do actually find history data *useful* for justifying and explaining their work. Much of our research in this dissertation operated on faith in the hypothesis that *if* data workers had easy access to history *then* they *would* find history useful. In the end, it was highly encouraging to finally see actual evidence in the Verdant Study in support of this claim.

In conclusion, we hope that others find this dissertation informative and useful groundwork for their own ideas in experiment support. We estimate that experimentation will only become more important over time as programming becomes more advanced. With further automation, more and more of a human's role may be to make decisions and weigh options for how data work *should* be done out of a set of promising options a machine (e.g., AutoML) can come up with. Understanding effective experimentation, and understanding which decision paths people take and why in data work, is an area that may benefit substantially more tool support than we have today.

References

A

Amershi, Saleema, and Cristina Conati. "Combining unsupervised and supervised classification to build user models for exploratory learning environments." *Journal of educational data mining* 1, no. 1 (2009): 18–71.

Arrieta, Alejandro Barredo, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García et al. "Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI." *Information Fusion* 58 (2020): 82–115.

B

Ziv Bar-Joseph, Georg K Gerber, Tong Ihn Lee, Nicola J Rinaldi, Jane Y Yoo, François Robert, D Benjamin Gordon, Ernest Fraenkel, Tommi S Jaakkola, Richard A Young, and others. 2003. "Computational discovery of gene modules and regulatory networks". *Nature Biotechnology* 21, 11 (2003), 1337–1342.

I. Bergström and A. F. Blackwell, "The practices of programming," in Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on, 2016, pp. 190–198.

Hugh Beyer and Karen Holtzblatt. 1997. *Contextual design: defining customer-centered systems*. Elsevier

Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. 1993. "Toolglass and magic lenses: the see-through interface". In Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 73–80..

J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in Proceedings of the 4th international workshop on End-user software engineering, 2008, pp. 1–5.

Braun, Virginia, and Victoria Clarke. "Thematic analysis." (2012).

A. R. Brown and A. Sorensen, "Interacting with generative music through live coding," *Contemp. Music Rev.*, vol. 28, no. 1, pp. 17–29, 2009.

R. J. Brunner and E. J. Kim, "Teaching Data Science," *Procedia Comput. Sci.*, vol. 80, pp. 1947–1956, Jan. 2016.

B. Buxton, *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann, 2010.

C

John M Carroll and Mary Beth Rosson. 1987. *Paradox of the active user*. The MIT Press.

Charmaz, Kathy. *Constructing grounded theory: A practical guide through qualitative analysis*. Sage, 2006.

- 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.
- Chaudhary, Kunal, Andrew Head, and Björn Hartmann. “*Jupyter’s Archive: Searchable Output Histories for Computational Notebooks*.” (2019).
- K. Cheung and J. Hunter, “*Provenance explorer--customized provenance views using semantic inferencing*,” in International Semantic Web Conference, 2006, pp. 215–227.
- Paul Clements and Linda Northrop. 2001. *Software product lines: Patterns and practice*. Boston, MA, EUA: Addison Wesley Longman Publishing Co (2001).
- 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.
- Juliet Corbin and Anselm Strauss. 1990. *Grounded Theory Research: Procedures, Canons and Evaluative Criteria*. Zeitschrift für Soziologie 19, 6: 515.

D

- Thomas H Davenport and DJ Patil. 2012. “*Data Scientist: The Sexiest Job of the 21st Century*”. Harvard business review 90 (2012), 70–76.
- S. B. Davidson and J. Freire, “*Provenance and scientific workflows: challenges and opportunities*,” in Proceedings of the 2008 ACM SIGMOD international conference on Management of data, 2008, pp. 1345–1350.

E

- Eick, Stephen G., Joseph L. Ste. en, and Eric E. Sumner. “*Seesoft-a tool for visualizing line oriented software statistics*.” IEEE Transactions on Software Engineering 18, no. 11 (1992): 957–968.

F

- Danyel Fisher, Badrish Chandramouli, Robert DeLine, Jonathan Goldstein, Andrei Aron, Mike Barnett, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. “*Tempe: an interactive data science environment for exploration of temporal and streaming data*.” Tech. Rep. MSR-TR-2014-148.

- Robert DeLine and Danyel Fisher. 2015. “*Supporting exploratory data analysis with live programming*”. In Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on. IEEE, 111–119.

- Ryohei Fujimaki. “*AutoML 2.0: Is The Data Scientist Obsolete?*”. Forbes Apr 7, 2020,10:10am

G

- Gebru, Timnit, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé III, and Kate Crawford. “*Datasheets for datasets*.” arXiv preprint arXiv:1803.09010 (2018).

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.

Samuel Gratzl. 2015. *IPython tutorial given as part of Visual Analytics winter term 2015/2016 at Johannes Kepler University*. Retrieved January 6, 2018 from
https://github.com/sgratzl/ipython-tutorial-VA2015/blob/master/03_Plotting.ipynb

T. Green, “*Programming Languages as Information Structures*,” in *Psychology of Programming*, J. M. Hoc, T. R. Green, R. Samurcay, and D. J. Gilmore, Eds. Academic Press, 1990, pp. 117–137.

T. R. G. Green and M. Petre, “*Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework*,” *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996.

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

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

H

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.

B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer, “*Design as exploration: creating interface alternatives through parallel authoring and runtime tuning*,” in Proceedings of the 21st annual ACM symposium on User interface software and technology, 2008, pp. 91–100.

Robert Hawley. 1987. *Artificial intelligence programming environments*. Intellect Books.

Head, Andrew, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. “*Managing messes in computational notebooks*.” In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–12. 2019.

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

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.

C. Hill, R. Bellamy, T. Erickson, and M. Burnett, “*Trials and tribulations of developers of intelligent systems: A field study*,” in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2016 IEEE Symposium on, 2016, pp. 162–170.

Hou, Youyang, and Dakuo Wang. "Hacking with NPOs: collaborative analytics and broker roles in civic data hackathons." Proceedings of the ACM on Human–Computer Interaction 1, no. CSCW (2017): 1–16.

Scott E. Hudson, Roy Rodenstein, and Ian Smith. 1997. "Debugging Lenses: A New Class of Transparent Tools for User Interface Debugging". In Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST '97), 179–187.

I

J

J. G. March, "Exploration and exploitation in organizational learning," *Organ. Sci.*, vol. 2, no. 1, pp. 71–87, 1991.

Jupyter Notebook 2015 UX Survey Results. Jupyter Project Github Repository. Retrieved Spring 2017 from

https://github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report_dashboard.ipynb

Jupyter Project. "A gallery of interesting Jupyter Notebooks." [Online]. Available: <https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks>. [Accessed: 24-Apr-2018].

K

Kandel, Sean, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank Van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono.

"Research directions in data wrangling: Visualizations and transformations for usable and credible data." *Information Visualization* 10, no. 4 (2011): 271–288.

C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in Proceedings of the SIGCHI conference on Human factors in computing systems, 2007, pp. 1455–1464.

M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting Exploratory Programming by Data Scientists," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2017.

Kim, Miryung, Thomas Zimmermann, Robert DeLine, and Andrew Begel. "Data scientists in software teams: State of the art and challenges." *IEEE Transactions on Software Engineering* 44, no. 11 (2017): 1024–1038.

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.

Donald Ervin Knuth. 1984. "Literate programming". *Computer Journal* 27, 2: 97–111.

Amy J Ko, Htet Htet Aung, and Brad A Myers. 2005. "Design requirements for more flexible structured editors from a study of programmers' text editing". In CHI'05 extended abstracts on human factors in computing systems. ACM, 1557–1560.

Amy. Ko and B. Myers, “Debugging reinvented,” in Software Engineering, 2008. ICSE’08. ACM/IEEE 30th International Conference on, 2008, pp. 301–310.

Ko, Amy J., and Brad A. Myers. "Designing the whyline: a debugging interface for asking questions about program behavior." In Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 151–158. 2004.

Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrence, Henry Lieberman, Brad Myers, and others. 2011. “The state of the art in end-user software engineering”. ACM Computing Surveys (CSUR) 43, 3 (2011), 21.

R. Koschke, R. Falke, and P. Frenzel, “Clone Detection Using Abstract Syntax Suffix Trees,” in 2006 13th Working Conference on Reverse Engineering, 2006.

S. K. Kuttal, A. Sarma, and G. Rothermel, “History repeats itself more easily when you log it: Versioning for mashups,” Proc. - 2011 IEEE Symp. Vis. Lang. Hum. Centric Comput. VL/HCC 2011, pp. 69–72, 2011.

Sandeep Kaur Kuttal, Anita Sarma, Amanda Swearngin, and Gregg Rothermel. 2011. “Versioning for Mashups—An Exploratory Study”. In the International Symposium on End User Development. Springer, 25–41.

L

T. D. LaToza and B. A. Myers, “Hard-to-answer questions about code,” in Evaluation and Usability of Programming Languages and Tools, 2010, p. 8.

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

Joseph Lawrence, Rachel Bellamy, Margaret Burnett, and Kyle Rector. 2008. “Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks”. in Proceedings of the twenty-sixth annual CHI conference on Human factors in computing systems – CHI ’08.

Lee, Angela, Doris Xin, Doris Lee, and Aditya Parameswaran. "Demystifying a dark art: Understanding real-world machine learning model development." arXiv preprint arXiv:2005.01520 (2020).

Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. “How software engineers use documentation: The state of the practice”. IEEE Software 20, 6: 35–39.

Liu, Yang, Tim Althoff, and Jeffrey Heer. "Paths explored, paths omitted, paths obscured: Decision points & selective reporting in end-to-end data analysis." In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, pp. 1–14. 2020.

Aran Lunzer and Kasper Hornbæk. 2008. Subjunctive interfaces: Extending applications to support parallel setup, viewing and control of alternative scenarios. ACM Transactions on Computer-Human Interaction (TOCHI) 14, 4 (2008), 17.

M

Macke, Stephen, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. "Fine-grained lineage for safer notebook interactions." arXiv preprint arXiv:2012.06981 (2020).

Robert C. Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Z. Merali, "Error: Why scientific programming does not compute," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

Mitchell, Margaret, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. "Model cards for model reporting." In Proceedings of the conference on fairness, accountability, and transparency, pp. 220–229. 2019.

Mittelstadt, Brent. "Principles alone cannot guarantee ethical AI." *Nature Machine Intelligence* 1, no. 11 (2019): 501–507.

N. Montfort, *Exploratory Programming for the Arts and Humanities*. MIT Press, 2016.

E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?". in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 1–11.

Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. "Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools," *IEEE Computer*, Special issue on UI Design, 49, issue 7, July, 2016, pp. 44–52.

N

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.

Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. "A survey of scientific software development." In Proceedings of the 2010 ACM–IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, 12.

Clemens Nylandsted Klokmose and Pär-Ola Zander. 2010. *Rethinking Laboratory Notebooks*. In Proceedings of COOP 2010. Springer, London, 119–139.

O

Gerard Oleksik, Natasa Milic-Frayling, and Rachel Jones. 2014. "Study of Electronic Lab Notebook Design and Practices That Emerged in a Collaborative Scientific Environment." In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '14), 120–133.

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.

P

- David Lorge Parnas. 1994. *Software aging*. In Proceedings of the 16th international conference on Software engineering, 279–287.
- Passi, Samir, and Steven J. Jackson. "Trust in data science: Collaboration, translation, and accountability in corporate data science projects." Proceedings of the ACM on Human-Computer Interaction 2, no. CSCW (2018): 1–28.
- Kayur Patel, James Fogarty, James A Landay, and Beverly Harrison. 2008. "Investigating statistical machine learning as a tool for software development." In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, 667–676.
- Kayur Patel. 2010. "Lowering the barrier to applying machine learning". In Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology, 355–358.
- Kayur Dushyant Patel. 2013. *Lowering the Barrier to Applying Machine Learning*. Ph.D. Dissertation.
- Alexandre Perez and Rui Abreu. 2014. "A diagnosis-based approach to software comprehension". In Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014.
- Fernando Pérez and Brian E Granger. 2007. "IPython: a system for interactive scientific computing". Computing in Science & Engineering 9, 3 (2007), 21–29.
- Fernando Perez and Brian E. Granger. 2015. "Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science". Project Jupyter Blog. Retrieved from <http://blog.jupyter.org/2015/07/07/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science/>
- 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.
- Pimentel, João Felipe, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. "A survey on collecting, managing, and analyzing provenance from scripts." ACM Computing Surveys (CSUR) 52, no. 3 (2019): 1–38.
- D Piorkowski, S D Fleming, C Scuffidi, M Burnett, I Kwan, A Z Henley, J Macbeth, C Hill, and A Horvath. 2015. "To fix or to learn? How production bias affects developers' information foraging during debugging". In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). 11–20.
- Piorkowski, David, Soya Park, April Yi Wang, Dakuo Wang, Michael Muller, and Felix Portnoy. "How AI developers overcome communication challenges in a multidisciplinary team: A case study." Proceedings of the ACM on Human-Computer Interaction 5, no. CSCW1 (2021): 1–25.
- Peter Pirolli. 2007. *Information Foraging Theory*. In Information Foraging Theory. 3–29.

R

- Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. “*Foraging Among an Overabundance of Similar Variants*”. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems. ACM, 3509–3521.
- C. Reas and B. Fry, *Processing: a programming handbook for visual designers and artists*, no. 6812. Mit Press, 2007.
- Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Schneiderman, Randy Pausch, Ted Selker, and Mike Eisenberg. 2005. “*Design principles for tools to support creative thinking*”. (2005).
- M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and others, “*Scratch: programming for all*,” Commun. ACM, vol. 52, no. 11, pp. 60–67, 2009.
- M. P. Robillard, “*What makes APIs hard to learn? Answers from developers*,” IEEE Softw., vol. 26, no. 6, 2009.
- M. B. Rosson and J. M. Carroll, “*Active programming strategies in reuse*,” in European Conference on Object-Oriented Programming, 1993, pp. 4–20.
- Adam Rule, Aurelien Tabard, and James Hollan. 2018. “*Exploration and Explanation in Computational Notebooks*”. In ACM CHI Conference on Human Factors in Computing Systems.

S

- Sanchez, Téo, Baptiste Caramiaux, Jules Françoise, Frédéric Bevilacqua, and Wendy E. Mackay. “*How do People Train a Machine? Strategies and (Mis) Understandings*.” Proceedings of the ACM on Human-Computer Interaction 5, no. CSCW1 (2021): 1–26.
- D. W. Sandberg, “*Smalltalk and exploratory programming*,” SIGPLAN Not., vol. 23, no. 10, pp. 85–92, 1988.
- Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. “*Ten simple rules for reproducible computational research*.” PLoS Comput. Biol. 9, 10 (2013), e1003285.
- Sambasivan, Nithya, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M. Aroyo. ““*Everyone wants to do the model work, not the data work*”: Data Cascades in High-Stakes AI.” In proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–15. 2021.
- Jeff Sauro. 2011. “*What is a Good Task-Completion Rate?*” <https://measuringu.com/task-completion/>. (2011). Accessed: 2019-1-4.
- Judith Segal. 2007. “*Some problems of professional end user developers*”. In IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007). IEEE, 111–118.
- Beau Sheil. 1983. “*Environments for exploratory programming*”. Datamation 29, 7 (1983), 131–144.

- B. Sheil, *Power tools for programmers*. Morgan Kaufmann Publishers Inc., 1986.
- Helen Shen and others. 2014. “*Interactive notebooks: Sharing the code.*” *Nature* 515, 7525 (2014), 151–152.
- J. Snoek, H. Larochelle, and R. P. Adams, “*Practical bayesian optimization of machine learning algorithms,*” in *Advances in neural information processing systems*, 2012, pp. 2951–2959.
- Jean-Luc R. Stevens, Marco Elver, and James A. Bednar. 2013. “*An automated and reproducible workflow for running and analyzing neural simulations using Lancet and IPython Notebook*”. *Frontiers in neuroinformatics* 7.

T

- 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)*, 569–578.
- Terrizzano, Ignacio G., Peter M. Schwarz, Mary Roth, and John E. Colino. “*Data Wrangling: The Challenging Journey from the Wild to the Lake.*” In *CIDR*. 2015.
- M. Terry, E. D. Mynatt, K. Nakakoji, and Y. Yamamoto, “*Variation in element and action: supporting simultaneous development of alternative solutions,*” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 711–718.
- Michael Toomim, Andrew Begel, and Susan L Graham. 2004. “*Managing duplicated code with linked editing*”. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 173–180.
- Edward R Tufte. 2006. *Beautiful evidence*. Vol. 1. Graphics Press Cheshire, CT.
- J. W. Tukey, “*Exploratory Data Analysis,*” *Analysis*, vol. 2, no. 1999, p. 688, 1977.

W

- Wang, April Yi, Anant Mittal, Christopher Brooks, and Steve Oney. “How data scientists use computational notebooks for real-time collaboration.” *Proceedings of the ACM on Human-Computer Interaction* 3, no. CSCW (2019): 1–30.
- Wang, April Yi, Zihan Wu, Christopher Brooks, and Steve Oney. “*Callisto: Capturing the “Why” by Connecting Conversations with Computational Narratives.*” In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–13. 2020.
- Wang, April Yi, Dakuo Wang, Jaimie Drozdzal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. “*Themisto: Towards Automated Documentation Generation in Computational Notebooks.*” arXiv preprint arXiv:2102.12592 (2021).
- Wang, Dakuo, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. “*Human-ai collaboration in data science: Exploring data scientists’ perceptions of automated ai.*” *Proceedings of the ACM on Human-Computer Interaction* 3, no. CSCW (2019): 1–24.

Pete Warden, “*Why the term “data science” is flawed but useful: Counterpoints to four common data science criticisms*”, O’Reilly Radar Data Newsletter, 3/2011
<http://radar.oreilly.com/2011/05/data-science-terminology.html>

Weinman, Nathaniel, Steven M. Drucker, Titus Barik, and Robert DeLine. “*Fork It: Supporting Stateful Alternatives in Computational Notebooks.*” In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–12. 2021.

Wiktionary. “provenance – Wiktionary.” [Online]. Available:
<https://en.wiktionary.org/wiki/provenance>. [Accessed: 22-Apr-2018].

Wiktionary. “verdant – Wiktionary.” [Online]. Available:
<https://en.wiktionary.org/wiki/verdant>. [Accessed: 22-Apr-2018].

Greg Wilson. 2006. “*Software carpentry: getting scientists to write better code by making them more productive*”. Computing in science & engineering 8, 6: 66–69.

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

Kim, Won, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. “*A taxonomy of dirty data.*” Data mining and knowledge discovery 7, no. 1 (2003): 81–99.

Wongsuphasawat, Kanit, Yang Liu, and Jeffrey Heer. “*Goals, process, and challenges of exploratory data analysis: an interview study.*” arXiv preprint arXiv:1911.00568 (2019).

X

Yihui Xie. 2014. “*knitr: a comprehensive tool for reproducible research in R*”. Implement Reprod Res 1: 20.

Xin, Doris, Litian Ma, Shuchen Song, and Aditya Parameswaran. “*How Developers Iterate on Machine Learning Workflows.*” In IDEA Workshop at KDD. 2018.

Xin, Doris, Eva Yiwei Wu, Doris Jung-Lin Lee, Niloufar Salehi, and Aditya Parameswaran. “*Whither AutoML? Understanding the Role of Automation in Machine Learning Workflows.*” In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–16. 2021.

Y

Yang, Qian, Justin Cranshaw, Saleema Amershi, Shamsi T. Iqbal, and Jaime Teevan. “*Sketching nlp: A case study of exploring the right things to design with language intelligence.*” In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–12. 2019.

YoungSeok Yoon and Brad A Myers. 2012. “*An exploratory study of backtracking strategies used by developers*”. In Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering. IEEE Press, 138–144.

Youngseok Yoon, Brad A. Myers, and Sebon Koo. 2013. “*Visualization of fine-grained code change history*”. In Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on, 119–126.

Y. S. Yoon and B. A. Myers, “*A longitudinal study of programmers’ backtracking*,” in Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on, 2014, pp. 101–108.

YoungSeok Yoon and Brad A Myers. 2015. “*Supporting selective undo in a code editor*”. In Proceedings of the 37th International Conference on Software Engineering–Volume 1. IEEE Press, 223–233.

Z

Zhang, Amy X., Michael Muller, and Dakuo Wang. “*How do data science workers collaborate? roles, workflows, and tools.*” Proceedings of the ACM on Human-Computer Interaction 4, no. CSCW1 (2020): 1–23.

Zhang, Ge, Mike A. Merrill, Yang Liu, Jeffrey Heer, and Tim Althoff. “*Coral: Code representation learning with weakly-supervised transformers for analyzing data analysis.*” arXiv preprint arXiv:2008.12828 (2020).

Zheng, Alice, and Amanda Casari. “*Feature engineering for machine learning: principles and techniques for data scientists.*” O'Reilly Media, Inc., 2018.

Appendices

Appendix A: Exploratory Programming Study Materials	221
INTERVIEW PROTOCOL	221
SURVEY QUESTIONS	224
Appendix B: Notebook Usage Study & Query Design Study Materials	232
STUDY SETUP	232
Screening	232
Design Activities & Interview	232
NOTEBOOK USAGE INTERVIEW	232
Supplies	232
Protocol	232
Required Topics	233
Possible Questions List	233
QUERY DESIGN EXERCISE	234
Appendix C: Variolite Usability Study Materials	236
STUDY SETUP	236
Research Questions	236
Controls	237
Study Protocol	237
Question format	237
USABILITY STUDY PROGRAMMING TASKS	238
Question set 1: Titanic	238
Question set 2: Animal Shelters	240
Feedback Survey	242
Appendix D: Jupytercon Scavenger Hunt Study Materials	244
STUDY SETUP	244
User testing goals	244
Observation goals	244
Protocol	245
Recording	245
TUTORIAL	245
SCAVENGER HUNT TASKS	252
PARTICIPANT TASK ASSIGNMENT	257

Appendix E: Classroom Deployment Pilot Materials	259
STUDY SETUP	259
Protocol	259
Study Intro Verbal Script	260
Homework Assignment	261
Appendix F: The Verdant Study Materials	265
STUDY PROTOCOL	265
Tutorial	268
Programming Task	279
Scavenger Hunt Questions	280
Thematic Analysis Qualitative Codes	292
Appendix G: Query Design Exercise Results	293

Appendix A: Exploratory Programming Study Materials

This study is described in Chapter 3

INTERVIEW PROTOCOL

Introduction

- Explain one person will mainly be talking and one person taking notes
- Tell the user that the session will be (optionally) recorded
- Explain that comments user makes may show up in published research
- Caution participants not to discuss other people in any way that could identify them, such as their name. Also ask participants to not discuss sensitive information about other people.
- Establish open-ended partnership

[Make introductions, thank them for participating in interview.] Today we're going to ask you some questions about your process for doing programming work, especially work that is exploratory. [From here on, “[the exploratory programming activity]” will indicate the specific kind of programming task they talk about, whether that be data analysis or some other kind of exploratory programming work]

[confirm permissions interviewee gave on the consent form, reiterate that their identity will not be revealed] Also, if there are certain projects that you specifically do not want to be shared with the public, we will be happy to accommodate that either by pausing video recording or making sure that any comments and images associated with the project will not show up in published research.

Also, you see here are some pieces of paper and a [pencil/pen] in front of you. [If this meeting is over the internet, ask them to either a) have a piece of paper in front of them to show on the screen or b) use the note-taking program of their preference over screen-share to show sketches] If any time during the interview it's easier for you to sketch a diagram or snippet of code, please feel free to do that. If we need clarification for anything during the interview, we may ask you to sketch something. This is completely optional. [confirm optional permissions interviewee gave on the consent form, reiterate that their identity will not be revealed]

We also ask you, for confidentiality, please avoid identifying any other person in this interview, particularly anything that could be used to identify that person, like their name. Please also avoid anything that's sensitive information about another person.

[If this rule is broken during the interview, interviewer will stop recording and rewind to erase that information. Remind the interviewee of this rule.]

Finally, you are in the driver's seat for this interview. You do not have to answer any questions that make you feel uncomfortable and you can choose not to perform any tasks that we ask you to do. Do you have any questions before we begin?

Background

How much of your time do you spend on [the exploratory programming activity]? How long have you been doing [the exploratory programming activity]? What tools do you use? How long have you been using these?

Stories

Can you tell us about some [the exploratory programming activity] that you've worked on recently? If you are alright with it, we would like you to show us some of your code and project pieces as we continue the interview, so that as you discuss your work practice, you can point out some concrete examples. In code you show us, I will also ask you questions about your coding practices and what motivates you to make certain coding decisions.

How long is your [the exploratory programming activity] process for a project? How much code do you write for this task? How long does it take to run?

Programming Practice

How much do you code using the console versus scripts?

How do you organize your [scripts or console, depending on response to previous question]? How is all the code you generate for this project organized into different files? Do you put multiple kinds of code tasks in one file? If so, how do you tell what code does what in the file?

How do you organize the output or results?

How often do you return to old code? How readable is this code later on?

How often do you share your code with someone else? Does this affect how you program?

How often do you communicate the incremental results of your work to someone else? How do you communicate this?

Exploratory Programming

Can you describe a time where you tried out several different avenues to produce an effective program? How often does this happen? How often do you try different possibilities while developing code? How often do you compare between two pieces of code? What do these comparisons look like?

Do you use version control? How do you manage versions of your code? Are there any challenges you've encountered with versioning, like wanting to return to an earlier version of the program?

How much of your work would you consider exploratory? What parts of your work are exploratory? How often do you work on exploratory programming?

Wrap Up

Do you have any thoughts on how the tools you use can be improved? We are interested in the [the exploratory programming activity] process, especially programming practices for exploring ideas. Can you think of ways to improve either or both?

Do you have anything else you think we should know about the [the exploratory programming activity] process?

Thank you for your time.

SURVEY QUESTIONS



This survey is part of a research study from the Natural Programming Group at the Human-Computer Interaction Institute at Carnegie Mellon University.

For any questions/concerns, please feel free to contact the research team:

Investigator: Mary Beth Kery mkery@cs.cmu.edu

Faculty Advisor: Prof. Brad Myers bam@cs.cmu.edu

The purpose of this survey is to understand your programming practices, when you are writing code to do exploratory data analysis. Here, “exploratory data analysis” includes asking questions of data, building models from data, writing code to filter, “clean” and/or visualize data, and data mining. “**Exploratory** data analysis” includes any computational-focused research where you are working with data to achieve some goal, but the exact means to that goal is unknown to you without trying different approaches and ways of manipulating data (and hence you must “explore” what code will work)..

Participation in this research is voluntary, and you have the right to withdraw from the survey at any time. Also, you must be at least 18 years old to participate. Your responses to this survey will be used in research: this means that your responses may be included with all our results from this survey in published research, but your responses will be de-identified first (we will remove any reference to your job, age, email, and remove other information that could be used to directly identify you). The sponsor of this research, the National Science Foundation, may access our research records including the full survey data. We will take caution to keep your information securely, and all identifying information confidential. That said, there is a risk of breach of confidentiality. There are no other risks outside those of normal internet browsing. There are no benefits either, besides gaining a better sense of your own practices.

At the end of the survey you will have the option to enter in a raffle for a \$25 Amazon gift card.

I am 18 years or older.

- Yes
- No (*will end the survey*)

I have read and understand the information above.

- Yes
- No (*will end the survey*)

I do, or have done exploratory data analysis as part of my job, and would like to participate in this survey.

- Yes (*will begin the survey*)
 - No (*will end the survey*)
-

Survey page 1

First, we will ask you some basic questions to get a sense of your work. The entire survey is 30 questions.

1. Gender
 - a. Male
 - b. Female
 - c. Other
 - d. Prefer not to say
2. Age
_____ years
3. What is your occupation?
4. How long have you been working in your field?
_____ years
5. When you work with data, how often does this require you to do *exploratory data analysis*?
 - a. Always
 - b. Very Often
 - c. Often
 - d. Occasionally
 - e. Rarely
 - f. Never [*survey will end with this option*]
6. Please describe your current or most recent project that involved some form of exploratory data analysis. If the details of your project are confidential, feel free to give only a brief high-level description of what you did:
[Open-ended text response _____]
7. When you work with data, how often do you *write code* to do this work?
 - a. Always
 - b. Very Often
 - c. Often
 - d. Occasionally
 - e. Rarely

- f. Never
8. How much formal training in computer science or related field (e.g., software engineering, information systems, etc.) do you have?
- None: self-taught
 - A few classes
 - A bachelor's major or minor in CS
 - Graduate degree

Survey page 2

Please answer all the following questions in this survey about projects you have done that involved exploratory data analysis.

How much do you agree or disagree with the following statement:

9. "I analyze a lot of different questions about the data in a single source code file"
- Strongly disagree
 - Disagree
 - Neither disagree nor agree
 - Agree
 - Strongly agree

How much do you agree or disagree with the following statement:

10. "When I am doing exploratory programming, I prioritize finding a solution over writing high-quality code"
- Strongly disagree
 - Disagree
 - Neither disagree nor agree
 - Agree
 - Strongly agree

11. Of the code you write for data analysis, what percent of all of that code is seen by other people? (Please estimate)

Percent of your code is only seen by you: ____ %

Percent of your code which is shared with your collaborators/team: ____ %

Percent of your code which is shared with the public (e.g., on an open source):

____ %

12. How often do you copy a file to make a new variant of that code?

- Always
- Very Often
- Often
- Occasionally
- Rarely

- f. Never
13. [If answered more frequently than Never] When you create such a file, how often do you name it based on the name of the original file (for example, you start with a file called “myfile.py” and then create a “myfile_v2.py” or “myfile_2016-July_9”)?
- g. Always
 - h. Very Often
 - i. Often
 - j. Occasionally
 - k. Rarely
 - l. Never
14. How often do you copy-and-paste pieces of code to reuse in different places?
- m. Always
 - n. Very Often
 - o. Often
 - p. Occasionally
 - q. Rarely
 - r. Never
15. How often do you keep old functions or snippets in your code that your analysis is *not* currently using?
- a. Always
 - b. Very Often
 - c. Often
 - d. Occasionally
 - e. Rarely
 - f. Never
16. How often do you comment out sections of code to make that code inactive?
- a. Always
 - b. Very Often
 - c. Often
 - d. Occasionally
 - e. Rarely
 - f. Never
17. [If answered more frequently than Never] What do you do with these commented out pieces of code? (check all that apply)
- Delete them once I decide I won’t need them anymore
 - Keep them in the code to keep track of what I have attempted
 - Keep them in the code in case I need to use them again
- What are some other reasons that you comment out code: _____

18. Check all of the following for how you *document* your code by writing comments during exploratory data analysis?

- I put comments in on most function/methods
- I put comments in most source code files
- I put comments for confusing parts of the program
- I add in comments when I'm sharing the source code
- I add in when I'm refactoring code (cleaning up the code)
- I add in comments occasionally when I remember to
- I pretty much never enter comments during exploratory data analysis
- Other: _____

19. Within the past 6 months, how often have you gone back to revisit or copy code from a previous project or a previous version in your current project?

- a. Never
- b. Once
- c. About twice
- d. About once a month
- e. About every two weeks
- f. About every week
- g. More often

20. When you are going back to look at previous code, what information might be useful in helping you find the right older code? (check all that apply)

- A preview of its output
- An image of any graphs it produced
- The time and date it was created
- The time and date it was last run
- The time and date of the last modification/edit
- Code diff between versions
- User-defined version number
- Comments or annotations you added
- Other: _____

When working on an exploratory data analysis project, do you ever run into issues with any of the following...

21. Distinguishing between similarly named versions of code files or output files

- a. Not at all a problem
- b. A minor problem
- c. A Significant problem
- d. A very big problem
- e. Cannot answer / don't know

22. Keeping track of your exploratory data analysis at a high-level; that is, what you have already tried or not tried for your analysis

- a. Not at all a problem
- b. A minor problem
- c. A Significant problem
- d. A very big problem
- e. Cannot answer / don't know

23. Keeping track of the data, scripts, or parameters that produced a particular result

- a. Not at all a problem
- b. A minor problem
- c. A Significant problem
- d. A very big problem
- e. Cannot answer / don't know

24. When you go back to a project you haven't looked at in a while, in general how easy is it to re-orient yourself with that project?

- a. Not at all a problem
- b. A minor problem
- c. A Significant problem
- d. A very big problem
- e. Cannot answer

25. When you are writing code to explore a research question, do you have any other difficulties or barriers besides what are listed above?

Survey page 3

Now, for the next few questions, please consider only one recent or current exploratory data analysis project. Ideally this would be a relatively large project, but still one you remember well.

26. On your recent project, what tools did you use? (select all that apply)

- Text Editors
- Console
- iPython Notebooks/Markdown
- Spreadsheets
- MatLab
- RStudio
- Other(s) _____

27. What programming languages were you working in for analysis on this recent project?
(select all that apply)

- R

- Python
- Fortran
- Ruby
- Scala
- Java
- C++
- Excel
- Google Spreadsheets
- Other(s) _____

28. For a project with exploratory data analysis, do you use any version control tool?

- Github
- SVN
- Bitbucket
- Other Version control tool: _____
- I don't use version control for this

29. [If they do Not use a version control tool above] If not, why? (check all that apply)

- I don't know what version control is
- I know what it is, but do not know how to use it
- Too complicated or too heavy-weight for what I need
- I don't collaborate
- I keep backups of files in a different way
- I am not concerned about reverting to older versions of my code
- Other _____

30. When you were developing code on your last project, how often did you need to revert to code you had before?

a. _____ times per project

Final Page

Thank you for participating in this survey!

If you are interested in being put into a raffle for a \$25 Amazon gift card, please provide your email address below.

[_____]

[] Optional: I am okay with researchers possibly contacting me by email about a follow-up interview that is part of a different, but related study.

[email if not provided above]

Finally (totally optional), if you have any feedback about this survey, feel free to let us know what you think!

[_____]

For any questions/concerns, please feel free to contact the research team:

Investigator: Mary Beth Kery mkery@cs.cmu.edu

Faculty Advisor: Prof. Brad Myers bam@cs.cmu.edu

Appendix B: Notebook Usage Study & Query Design Study Materials

Both of these studies are described in Chapter 4, and were run at the same time and setting

STUDY SETUP

Jupytercon 2017 industry/researcher conference for users of the Jupyter tools ecosystem. This study was conducted at one table in the User Study Room, hosted by Project Jupyter and Bloomberg L.P. at the conference. All participants were conference attendee walk-ins, with the exception of the Query Design Exercise, which additionally was open as a survey online to remote participants during the duration of the conference.

Screening

Screening check [said aloud]: This table is recruiting people who actively use notebooks for doing data work.

Design Activities & Interview

We set up stations at a table, with different activities going on at the same time:

- Design Activity: Query Exercise
- Notebook Usage Interview Study

After completing part 0 and part 2, a participant was invited to take a seat at any open station. Participants were encouraged to do all stations at our table, but could complete as many as they wished to or had time for.

NOTEBOOK USAGE INTERVIEW

Supplies

- Audio recorder device
- Paper consent forms & pens
- Experimenter's laptop for note taking during the interview

Protocol

1. Participant reads through and signs a paper consent form.
2. Interviewer asks participant to give optional oral consent for their interview to be recorded. If the participant agrees, the Interviewer starts audio recording. If not, the Interviewer opens an empty notes doc to type interview notes.

3. Interviewer asks participant: If you have a laptop on you, and are willing to pull up a past significant data project on your computer, please do.
4. Interviewer asks the opening question: “*First, can you briefly describe what you use Jupyter notebooks for?*”
5. Depending on participant’s answers and time limit, the interviewer conducts open-ended conversation touching upon required topics, and any questions from the question list, both shown below.

Required Topics

- What kinds of things do you use Jupyter notebooks for?
- How long, in terms of hours, days, or weeks, is your typical project?
- In your Jupyter notebook, how do you deal with history?
- For what kind of project do you care about keeping older versions of your code or older output? For what kind of projects do you not?
- How do you currently keep any old experiment information?
- Do you use version control?

Possible Questions List

1. (data science practice) Do you collaborate in your data science work, if so how? Do you collaborate using notebooks in particular, and if so how?
2. (literate programming) How literate is literate programming ie how much do people write descriptions or break up their analyses into small pieces? (probably it depends)
3. (literate programming) How messy do notebooks get? Does the block organization lead to inherently more cleanly structured code than what people do without the notebooks? Or not?
4. (literate programming) How do people debug in a literate programming context and is it really any different from debugging a plain program?
5. (literate programming) How easy is it to change around functions or restructure a notebook as compared to a normal program? For example, the block structure, while it has benefits for organization, might make changing the program harder.
6. (literate programming) How is literate programming used in different contexts, and for different purposes?
7. How does exploratory programming fit in with your work practices?
8. How much exploratory programming do you do using Jupyter notebooks or some other tools instead?
9. How do you deal with versioning with Jupyter notebooks?
10. How long (something similar to lines of code) are your notebooks? How many notebooks do you typically have?
11. How often do you share your notebooks?
12. At what stage of “polish” or how much effort do you take in organizing/adding notes to your notebook before sharing it or showing it to others?
13. How do you handle an iterative process in a notebook?
14. What do you not do in notebooks? What do you think they’re not great for and why?

15. While working on a notebook, how much effort do you put into note-taking or organization? Why?
16. When you were a novice doing data science for X what was the hardest thing to learn about programming for data science?
17. How often do you use version control? Do you need it? Why or why not.
18. How often do you consult or copy examples from other people's notebooks? Other people's code?
19. How do you keep track of the things you try? Do you need to?
20. What is currently the most challenging thing for you as a data scientist? With programming tools?
21. What programming languages/tools do you use and why? Is this your company's choice or your own?
22. How often do you share in-progress data science work with someone else? Why?
23. How long do you typically use a given notebook for?
24. How often do you refer to old notebooks or adapt content from prior notebooks?
25. Before notebooks what did you use? Are there any parts of your prior practice that you preferred to what you do now?
26. What do you do to understand your data? How does programming play a role in that? How often are you visualizing your data?
27. What are you doing with data? How long does it take you to achieve something? Are you doing short simple analyses, like visualizing a distribution, or something more long and complex? How do notebooks compare for short simple analyses versus long complex tasks?
28. Is your analysis later incorporated into some kind of software system? If so, how do you transition from exploratory analyses in notebooks to a more formal software engineering setting?

QUERY DESIGN EXERCISE

The following is the survey delivered to participants both in person (on a laptop) and remotely via the internet:

Journey to the past (of your data work)

The purpose of this survey is to understand your programming practices, when you are writing code to do exploratory data analysis. Here, “exploratory data analysis” includes asking questions of data, building models from data, writing code to filter, “clean” and/or visualize data, and data mining. “Exploratory data analysis” includes any computational-focused research where you are working with data to achieve some goal, but the exact means to that goal is unknown to you without trying different approaches and ways of manipulating data (and hence you must “explore” what code will work)..

To participate in this study you must be at least 18 years old.

As described above, do you do exploratory data analysis as part of your job and are at least 18?
[yes, no --- no ends survey]

----- *next page* -----

Take a moment to recall a data project that you did recently.

Briefly, what was this project about?

[long answer text entry]

----- *next page* -----

Imagine you have a magical perfect record of every analysis run you did in this project. You also have a magic search engine that can retrieve you any code version, parameters used or output from the past.

What would you like to type in to find an item from the past?

[short answer text entry]

Given your own work practices, type as many queries as you can think of that could be helpful to you to retrieve a past experiment. Don't worry about feasibility. Phrase it in natural human language like you're talking to a colleague.

[long answer text entry]

----- *next page* -----

Has **not** being able to find a past experiment ever caused you problems? Is yes, what happened?

[long answer text entry]

In a project, how often do you want to look back at prior experiments you've done?

[Likert scale: Never ---> Very Often]

----- *next page* -----

Thank you wonderful human! If you'd be willing for us researchers from Jupyter or Carnegie Mellon University to possibly interview you later about your data science practices, please enter your email.

[short answer text entry]

Appendix C: Variolite Usability Study Materials

This study is described in Chapter 7

STUDY SETUP

The goal of this study is to test the usability of variant box interactions for managing *in-situ* versioning of code. We include a control condition where participants have no extra tool support to compare efficacy and timing.

Conditions 4x4 matrix:

	Tool condition 1st	Control condition 1st
Titanic questions 1st	Tool/Titanic, Control/Animal-Shelter	Control/Titanic, Tool/Animal-Shelter
Animal-Shelter questions 1st	Tool/Animal-Shelter, Control/Titanic	Control/Animal-Shelter, Tool/Titanic

Control/Animal-Shelter, Tool/Titanic Condition 1	P1, P5, P9, P13
Tool/Animal-Shelter, Control/Titanic Condition 2	P2, P6, P10, P14
Control/Titanic, Tool/Animal-Shelter Condition 3	P3, P7, P11, P15
Tool/Titanic, Control/Animal-Shelter Condition 4	P4, P8, P12, P16

Research Questions

RQ1: Can people successfully use variant boxes? What kinds of barriers do they encounter?

RQ2: Can people more successfully (finding section, then acting upon that code), compared to anything they might try, return to previous versions in order to

- a) Read the code
- b) Run as is
- c) Edit into a new version

Controls

- Two datasets of similar kinds of data.
- We shortened the animal shelter dataset to match the number of rows of the titanic dataset and shortened the number of columns in the titanic dataset to match the number of columns of data in the animal shelter dataset. The importance of having each dataset equivalent size is A) time participants needs to look over the dataset (#columns) and B) time their program takes to run on that data (#rows)
- Tasks of equivalent format and difficulty across conditions

Study Protocol

Time total: 90 minutes approximately = \$15 pay

1. Consent form and study setup (10 minutes)
2. Tutorial for Condition 1 (5 minutes) & Tutorial where we show them the terminal and how to run Python (5 minutes)
3. Condition 1 questions (30 minutes)
4. Tutorial for Condition 2 (5 minutes)
5. Condition 2 questions (30 minutes)
6. Post-study survey (10 minutes)

Question format

In each question set, the participant will have a series of 20 questions. Questions will be put in Google forms so that they are delivered 1 by 1. The reason for this is that some later questions will ask the participant to backtrack to previous tasks and change them to ask something slightly different.

For each question set, the participant will be given 1) the Atom editor open, 2) a folder with the dataset, and 3) a starter script that loads in the data and explains what all the columns in the dataset are. The reason to provide them with a starter script is to save time figuring out how to properly load the data, and to get them to the stage of exploratory data analysis more quickly.

The drawback is, if they are unfamiliar with the libraries we use to load in data, this may throw them.... Though participants are free to modify the starter script to use whatever libraries they are used to.

USABILITY STUDY PROGRAMMING TASKS

Participants completed one of two possible question sets.

Question set 1: Titanic

The Titanic is a famous ship that sank on its first voyage in 1912, killing over half of those on board. One reason for so few survivors was that there were not enough lifeboats on the ship to hold all passengers and crew. You are on a team of marine archaeologists studying the Titanic. The team has asked you to do some data analysis on the ship records to help them better interpret their archaeological findings from this shipwreck at the bottom of the ocean.

In each task, you will have a set amount of time. Your goal is to **answer each question as quickly as possible**. Once you have an answer, ask the experimenter if your answer is correct. (30 minutes total)

(Each task is 5 minutes. As soon as the person gets an answer, they can get the experimenter to tell them if it is correct, and if it is not, try to correct it within the 5 minutes. If the person gets stuck and is not able to get a correct answer after 5 minutes, the experimenter will try to help them get the right answer to move on?)

You have just joined a team of marine archaeologists as an intern. A previous intern began a number of data analyses for the team, and your boss has asked you to use the old intern's code and pick up where they left off.

----- next page -----

You may notice in the old code there are calls to a library underwaterViz. This library will help divers visualize data on underwater helmet screens to direct their search.... but really, just ignore this library! Please don't delete the old intern's code that uses it, but you don't need to add it to your code.

1. [RQ2 a & c] In titanic.py, change the marriageStats() function to instead count the survival rate of BOTH married women and married men. Your boss no longer cares if the passenger's husband/wife was actually onboard or not, so please count any married person.
2. [RQ2 a & c] Did children on the Titanic with more siblings fare better? Please change the function familySizeStats() to count only child passengers (under the age 18.) Also, please don't count the child's parents (the "Parch" field) in the total family size count.
3. [RQ2 b] Oops. --- Your boss realized he's forgot to record the answer to "What portion of married women survived the Titanic who also had spouses onboard?". Please go back to marriageStats() original code and run it again to get the answer.

4. [RQ2 b] ...Well of course you boss also forgot to record the answer to "What was the survival rate of passengers by their family size?". Please go back to familySizeStats() original code and run it again to get the answer.
5. [RQ2 a & c] We know that more women and children survived the Titanic. Did having a larger family improve the survival likelihood of men too, or just women and children? Change familySizeStats() to count the ratio of adult men that survived per each family size.
6. [RQ2 a & c] Now, how many married women or married men were on board who were also traveling with children? (Estimate the "Parch" field in the data to be the total number of children that passenger had.) Please change marriageStats() to answer this question when it is called.
7. Your team notes that some passengers were French. "Mrs" in french is "Mme" and "Mr" in french is "M". Please give the team new answers to all the previous marriage related questions you answered for the team to count french passengers. (portion of married women that survived, portion of married men/women that survived, portion of married women with children that survived)

----- next page -----

1. What percentage of female passengers survived the titanic? (5 minutes)
Pilot time: ??
2. What percentage of passengers of each class survived the titanic? (5 minutes)
Pilot time: 7 minutes
3. (*go back, overlapping*) Out of all female passengers *under the age of 20*, what percentage survived the titanic? (5 minutes)
Pilot time: 1:30 minutes
4. (*go back, overlapping*) Now, out of all child passengers (passengers under the age of 18), what percentage survived the Titanic from each class? Eg. 90% from 1st class, 90% from 2nd class, 90% from 3rd class (5 minutes)
Pilot time: 5:30 minutes
5. (*overlapping*) Consider the age ranges *child*: 0-17, *young adult*: 18-26, *adult*: 27-45, *middle-aged*: 46-65, and *elder*: 65+. For each age group, find the percentage of passengers who survived of that age group. (5 minutes)
Pilot time: 7:30 minutes

----- next page -----

Comprehension question:

Look at a Titanic analysis file, that either uses the tool or does not use the tool, depending on the condition.

6. Another member of your team has been analyzing different questions to do with the titanic. One question they answered was “*How many married women on the titanic survived?*”. However, your team noticed an issue with their analysis. They only counted passengers with the title “Mrs.” as married women, but several French passengers would have used the title “Mme” which would identify them as married women. Your team member is on vacation! Please fix their code to give the correct answer to “How many married women on the titanic survived”, this time counting both “Mrs” and “Mme”. (5 minutes)

Pilot time: 2:00 minutes

Question set 2: Animal Shelters

This dataset is from the Austin Animal Center, a large animal shelter in Texas. Shelter workers would like to better understand trends in animal adoption, in order to better help those pets who are less likely to find homes.

----- next page -----

In each task, you will have a set amount of time. Your goal is to **answer each question as quickly as possible**. Once you have an answer, ask the experimenter if your answer is correct. (30 minutes total)

1. [RQ2 a & c] In shelter.py, change the catsAdoptedStats() function to count BOTH dogs and cats. Also, the shelter workers point out that while adoption is ideal, transfer to a partner facility is also a good outcome. Change the catsAdoptedStats() function to count cats and dogs that are adopted or transferred (in Outcome field of data) to a partner (in OutcomeSubtype field of data).
2. [RQ2 a & c] It is a popular belief that pets with black fur are adopted less often than pets with any other color fur. With cats, this may be because of a superstition that black cats bring bad luck. Change the furColorStats() function to give the adoption percentage of black cats (whose fur color is simply “Black”) and ALSO the adoption percentage of all non-black cats so that shelter workers can see if this belief holds in this dataset.
3. [RQ2 a & c] Shelter workers note that animals in the dataset are described by different related colors. So, black cats may include the fur color “Black”, but also the fur colors like “Black Smoke” and any fur color that includes the word black. Revise furColorStats() so that it counts these different color variants as “Black”.
4. [RQ2 b] Shelter workers are interested in your findings! But, they would like you to move the analysis on black cats to a new function blackCatStats() and return furColorStats() to what it originally: the adoption rate of animals of each individual color.
5. [RQ2 b] Oh, since shelter workers are now discussing cats, they would also like the answer to “What was the adoption-only rate of cats in general?”. Please go back to catsAdoptedStats()

6. [RQ2 a & c] Now, what was the adoption-only rate of cats who were neutered, versus those who were not? Revise catsAdoptedStats() to give the adoption percentage of neutered cats (“Neutered Male” or “Spayed Female” in the data) and the adoption percentage of non-neutered cats (“Intact Male” or “Intact Female” in the data)
7. The shelter workers have been arguing about what a “good outcome” is, but now they’ve definitely decided that Adoption OR Transfer to a Partner are good outcomes they’d like to count pretty much as equivalent to adoption. Please update all the answers you’ve given them about fur color to count the pets transferred to a partner, not just adopted. (Questions are: Adoption rate by fur color, Adoption rate of cats with plain “Black” fur versus non-black cats, Adoption rate of cats with any type of “Black” fur versus non-black cats)

----- next page -----

1. What percentage of cats were adopted/transferred? (5 minutes)
Pilot time:
2. Some pets in the data are named, some not. Were pets who were named more or less likely to be adopted? (5 minutes)
Pilot time: 2:00 minutes
3. What percentage of cats were *adopted* into Foster care? (5 minutes)
Pilot time: 1:30 minutes
4. Out of all cats that were *adopted* (in general), what percentage of these cats were neutered? (“*Neutered*” or “*Spayed*” means *neutered*, while “*Intact*” means *not neutered*) (5 minutes)
Pilot time: 3:15 minutes
5. In the dataset, there are some animals that were euthanized due to rabies. Most household pets are given yearly rabies vaccines to prevent this dangerous disease. One possibility is that perhaps these animals that got rabies were feral or neglected, and may have been missing other normal medical procedures. Out of all the pets who were a “Rabies risk” (meaning they had rabies symptoms), what percentage of these pets were neutered? (5 minutes)
Pilot time: 2:00 minutes

----- next page -----

Comprehension question:

Look at a shelter analysis file, that either uses the tool or does not use the tool, depending on the condition.

6. An intern worked on this data analysis before you started on this project. Previously, they looked at how often dogs with black colored fur are adopted, compared to how often a dog with *any* fur color is adopted. There is a common superstition that black cats are unlucky, and shelter

workers would like to know: are black cats less likely to be adopted? Please change the intern's previous analysis to answer this question about cats. (5 minutes)

Feedback Survey

Your age?

[*short text answer*]

Your gender:

[*male, female, other, prefer not to say*]

Your occupation? If student, please say "masters student in history" or "undergraduate in bio" etc.

[*short text answer*]

How many years of programming experience do you have?

[*short text answer*]

How many years of experience do you have programming to do data analysis or machine learning?

[*short text answer*]

How often do you program to work on data analysis?

[*Several times a day, Several times a week, Every few months, For occasional projects, Never*]

When you do data analysis, how often is that exploratory data analysis?

[*Always, Very Often, Often, Sometimes, Rarely, Never, I don't typically work on data analysis*]

How familiar are you with a software version control system? (For example Git, SVN, or Mercurial)

[*Very Familiar, Somewhat Familiar, Somewhat Unfamiliar, I don't know how to use software version control, I don't really know what software version control is*]

----- *next page* -----

Tool Feedback

Please rate the tool you used

Ease of learning how to use it:

[*Very difficult ---> Very Easy*]

Overall, how did you like the tool?

[*Strong Dislike ---> Strong Like*]

Is there any particular feature you liked in the tool? Why?

[*Long text answer*]

Is there any particular feature you disliked like in the tool? Why?

[*Long text answer*]

If you were to use this tool in real life, what other things should the tool be capable of to meet how you would want to use it?

[*short text answer*]

Would you consider using it in real life? Why or Why not?

[*long text answer*]

Finally, do you have any other feedback about the tool or the study itself?

[*long text answer*]

Appendix D: Jupytercon Scavenger Hunt Study Materials

This study is described in Chapter 11

STUDY SETUP

Jupytercon 2018 industry/researcher conference for users of the Jupyter tools ecosystem. This study was conducted at one table in the User Study Room, hosted by Project Jupyter and Bloomberg L.P. at the conference. All participants were conference attendee walk-ins.

User testing goals

We want to be able to measure how effectively (if at all) a user can get back to a past state. We want design feedback to improve the tool. This is the 2nd user study of this tool. The 1st was a very basic click-through on an early prototype for potential adoption feedback and design feedback.

1. Is the tool sufficient to allow a user to get back to prior states and do so effectively?
2. Do the users understand the various visualizations, and/or can we get some actionable feedback on how to improve them?
3. Are users of the tool able to effectively use the various interactions (checkpoint list, filters, ghost book, inspector) together, or do they really only use one part of the tool at a time?
4. Which parts of the tool, if any, do the users go to most often to try to find a past state, when they are free to use any of the interactions?
5. Are there any parts of the tool that users just don't notice or don't think to use?

Observation goals

1. What path does the user follow, what series of states in the tool do they go to before (if) getting to the correct destination?
2. Are there any confusions that the user voices, where they are not sure how to interpret something they see on the screen?
3. When given free choice of where in the tool to start looking for something, where is the first state they go to?
4. In open-ended feedback, how do users feel about the tool? Do they see themselves adopting something like this, why not, or if so, in what situations?

Protocol

Participants complete all steps on a provided mouse, keyboard, and large monitor connected to a Macbook Pro. Open-ended feedback is captured using an audio recording device, if the user is comfortable with recording, otherwise the interviewer takes notes on another laptop.

1. Participant will read over a consent form <link> and sign. At the end of the form, they will also enter their secret code to link the data back.
2. Participant will get a brief introduction to the tool they're going to user test (standardized as a html page with a set of GIFs)
3. Participant gets instructions to think-aloud
4. Participant gets the first finding task. Each task is a scavenger hunt task on a piece of paper, which will first be read aloud by the experimenter and then be given to the participant as reference
5. After each finding task, the participant can choose to do another task or not
6. Finally, ask them for open-ended feedback on the tool

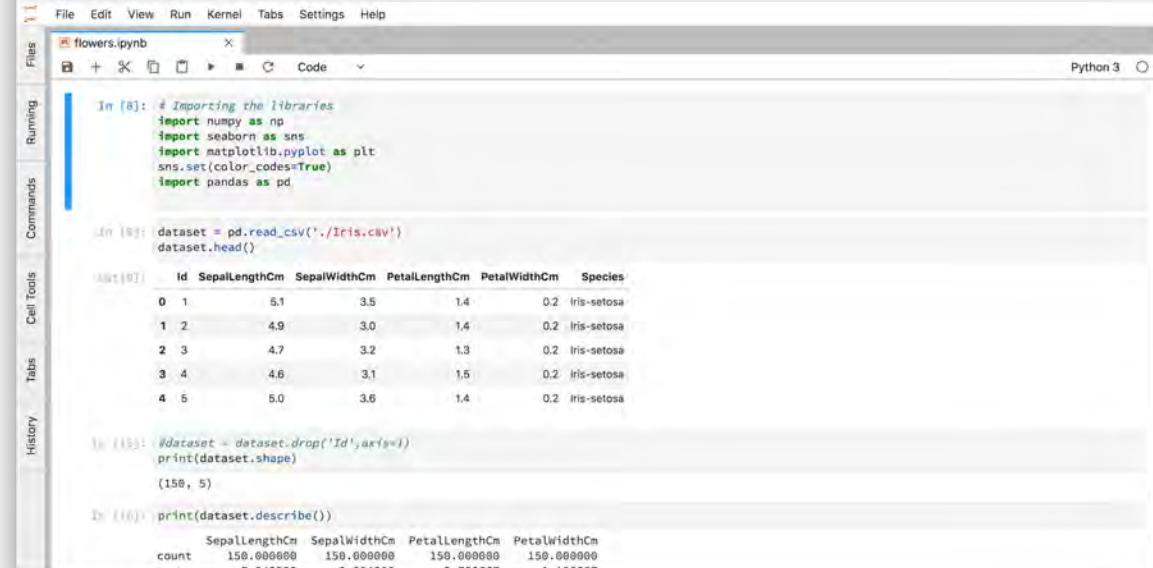
Recording

Screen recording, optional audio recording according to participant's comfort

TUTORIAL

Verdant is an extension for JupyterLab

It is an experimental prototype tool for recording and showing history back to the user of their work in the notebook



The screenshot shows a Jupyter Notebook interface with the following code execution history:

```

In [8]: # Importing the libraries
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(color_codes=True)
import pandas as pd

In [9]: dataset = pd.read_csv('./Iris.csv')
dataset.head()

Out[9]:
   Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0  1          5.1          3.5          1.4          0.2  Iris-setosa
1  2          4.9          3.0          1.4          0.2  Iris-setosa
2  3          4.7          3.2          1.3          0.2  Iris-setosa
3  4          4.6          3.1          1.5          0.2  Iris-setosa
4  5          5.0          3.6          1.4          0.2  Iris-setosa

In [10]: #dataset = dataset.drop('Id',axis=1)
print(dataset.shape)

Out[10]: (150, 5)

In [11]: print(dataset.describe())

```

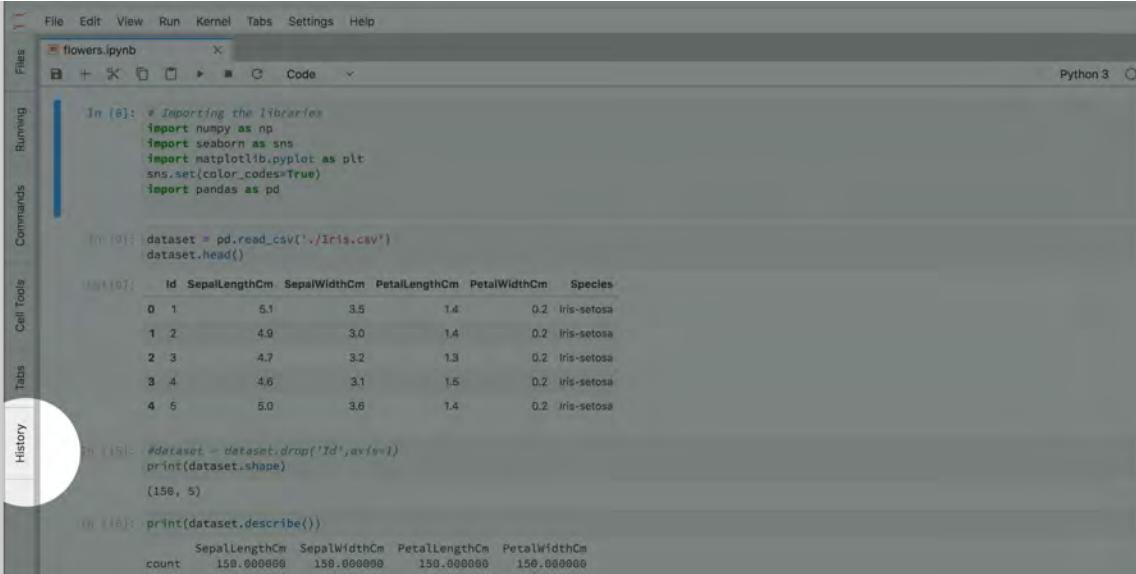
Cell 8: Importing libraries (numpy, seaborn, matplotlib, pandas) and setting seaborn color codes.

Cell 9: Reading the Iris dataset from a CSV file and displaying the first 5 rows.

Cell 10: Dropping the 'Id' column and printing the shape of the dataset.

Cell 11: Describing the dataset to show summary statistics for each column.

open Verdant with the history button



The screenshot shows a Jupyter Notebook interface with the 'History' tab highlighted by a white circle. The code execution history is identical to the one in the previous screenshot:

```

In [8]: # Importing the libraries
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(color_codes=True)
import pandas as pd

In [9]: dataset = pd.read_csv('./Iris.csv')
dataset.head()

Out[9]:
   Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0  1          5.1          3.5          1.4          0.2  Iris-setosa
1  2          4.9          3.0          1.4          0.2  Iris-setosa
2  3          4.7          3.2          1.3          0.2  Iris-setosa
3  4          4.6          3.1          1.5          0.2  Iris-setosa
4  5          5.0          3.6          1.4          0.2  Iris-setosa

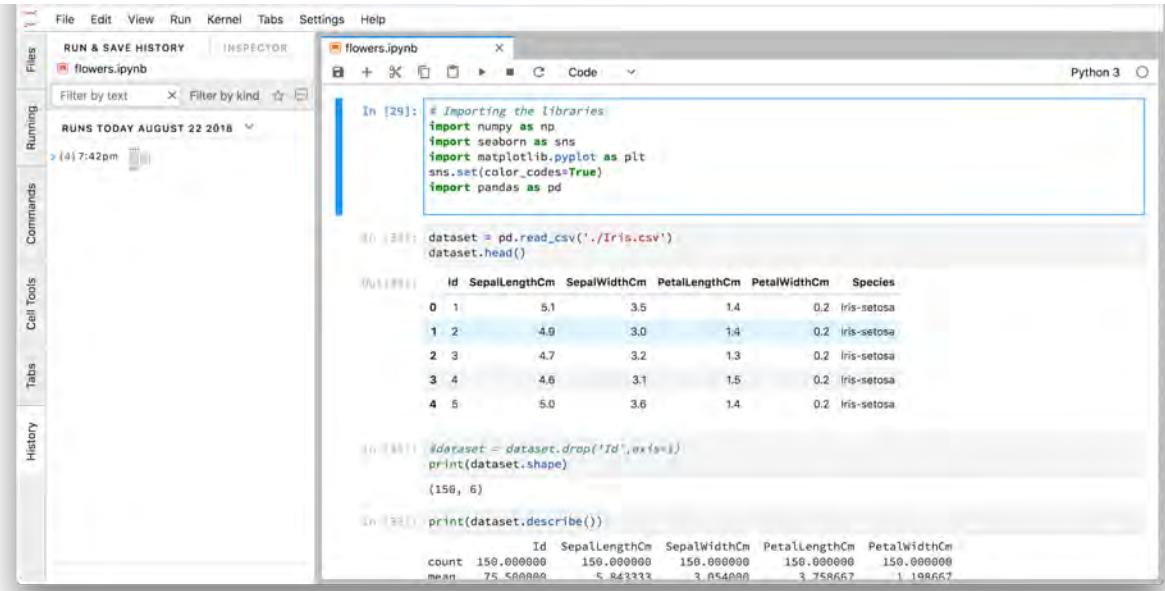
In [10]: #dataset = dataset.drop('Id',axis=1)
print(dataset.shape)

Out[10]: (150, 5)

In [11]: print(dataset.describe())

```

The history panel appears



A screenshot of a Jupyter Notebook interface. On the left, there's a vertical sidebar with tabs for File, Edit, View, Run, Kernel, Tabs, Settings, Help, Files, Running, Commands, Cell Tools, Tabs, and History. The History tab is currently selected. In the main area, there's a notebook titled "flowers.ipynb". The notebook contains several code cells and their outputs. The first cell (In [29]) shows code for importing libraries like numpy, seaborn, matplotlib, and pandas. The second cell (In [30]) shows the head of a dataset. The third cell (In [31]) drops the 'Id' column and prints the dataset's shape. The fourth cell (In [32]) prints the dataset's description, showing statistics for each column.

```
# Importing the Libraries
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(color_codes=True)
import pandas as pd

dataset = pd.read_csv('./Iris.csv')
dataset.head()

dataset = dataset.drop('Id',axis=1)
print(dataset.shape)

(150, 6)

print(dataset.describe())
   Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
count 150.000000 150.000000 150.000000 150.000000 150.000000
mean  75.000000  5.842222  3.754000  1.750000  0.800000
```

This shows the last 4 runs of
the notebook

Each vertical dash represents a cell in the notebook. The matchstick-looking ones are the cells that were run.

The screenshot shows a Jupyter Notebook interface with a sidebar on the left containing 'Running' and 'History' tabs. A tooltip over the 'Running' tab displays the text 'RUNS TODAY AUGUST 22 2018'. The main area shows a code cell (In [29]) with Python code for importing libraries and reading a CSV file. Below the code, the dataset's head and describe methods are called, resulting in tables of numerical and categorical data.

```

# Importing the libraries
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(color_codes=True)
import pandas as pd

dataset = pd.read_csv('iris.csv')
dataset.head()

      Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0   1       5.1          3.5         1.4        0.2  Iris-setosa
1   2       4.9          3.0         1.4        0.2  Iris-setosa
2   3       4.7          3.2         1.3        0.2  Iris-setosa
3   4       4.6          3.1         1.5        0.2  Iris-setosa
4   5       5.0          3.6         1.4        0.2  Iris-setosa

#dataset = dataset.drop('Id',axis=1)
print(dataset.shape)
(150, 6)

print(dataset.describe())

```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
count	150.000000	150.000000	150.000000	150.000000	150.000000	
mean	75.500000	5.843333	3.756667	1.756667	1.198667	

Open the run listing for more detail

This previews what the code and output looked like at the time when they were run

The screenshot shows the same Jupyter Notebook interface as above, but with a expanded 'Running' section in the sidebar. This section lists the individual runs, including the timestamp '7:42pm' and a 'make a note' button. The tooltip for the first run shows the expanded code and output, including the full 'dataset.describe()' output and the printed dataset head.

```

# Importing the libraries
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(color_codes=True)
import pandas as pd

dataset = pd.read_csv('iris.csv')
dataset.head()

      Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0   1       5.1          3.5         1.4        0.2  Iris-setosa
1   2       4.9          3.0         1.4        0.2  Iris-setosa
2   3       4.7          3.2         1.3        0.2  Iris-setosa
3   4       4.6          3.1         1.5        0.2  Iris-setosa
4   5       5.0          3.6         1.4        0.2  Iris-setosa

#dataset = dataset.drop('Id',axis=1)
print(dataset.shape)
(150, 6)

print(dataset.describe())

```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
count	150.000000	150.000000	150.000000	150.000000	150.000000	
mean	75.500000	5.843333	3.756667	1.756667	1.198667	

Click on code preview...

for yet more detail about this cell's history

The screenshot shows a Jupyter Notebook interface with the 'Inspector' tab active. The 'Inspector' pane on the left lists operations like 'make a note' and 'dataset = dataset.drop('Id', axis=1)'. The main area shows code and its output. In the code editor, a cell containing `#dataset = dataset.drop('Id', axis=1)` is highlighted. Its output cell, In [41]:, also contains the same code and a data frame:

```
In [41]: #dataset = dataset.drop('Id', axis=1)
print(dataset.shape)
dataset.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

This takes us to the inspector pane

The inspector gives a full list of the different versions a cell. Clicking around the notebook “inspects” the cells you click

The screenshot shows the Jupyter Notebook interface with the 'INSPECTOR' tab selected. The 'Running' section lists three previous runs (v3, v2, v1) with their respective code snippets and execution times. The 'Commands' section shows the history of commands run. The main notebook area displays the output of In[45] and Out[45], which are identical to In[44] and Out[44]. A large black arrow points from the text 'The inspector shows output history too' below to the Out[45] cell in the notebook.

```

In [44]: dataset = pd.read_csv('../Iris.csv')
dataset.head()

Out[44]:
   id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0   1           5.1          3.5           1.4          0.2  Iris-setosa
1   2           4.9          3.0           1.4          0.2  Iris-setosa
2   3           4.7          3.2           1.3          0.2  Iris-setosa
3   4           4.6          3.1           1.5          0.2  Iris-setosa
4   5           5.0          3.6           1.4          0.2  Iris-setosa

In [45]: #dataset = dataset.drop('Id',axis=1)
print(dataset.shape)
dataset.head()

Out[45]:
   id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0   1           5.1          3.5           1.4          0.2  Iris-setosa
1   2           4.9          3.0           1.4          0.2  Iris-setosa
2   3           4.7          3.2           1.3          0.2  Iris-setosa
3   4           4.6          3.1           1.5          0.2  Iris-setosa
4   5           5.0          3.6           1.4          0.2  Iris-setosa

```

The inspector shows output history too

If you click on output, its hististory will show up listed in the inspector as well

The screenshot shows the Jupyter Notebook interface with the 'INSPECTOR' tab selected. The 'Running' section lists three previous runs (v3, v2, v1) with their respective code snippets and execution times. The 'Commands' section shows the history of commands run. The main notebook area displays the output of In[45] and Out[45], which are identical to In[44] and Out[44]. A large black arrow points from the text 'The inspector shows output history too' above to the Out[45] cell in the notebook.

```

In [44]: dataset = pd.read_csv('../Iris.csv')
dataset.head()

Out[44]:
   id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0   1           5.1          3.5           1.4          0.2  Iris-setosa
1   2           4.9          3.0           1.4          0.2  Iris-setosa
2   3           4.7          3.2           1.3          0.2  Iris-setosa
3   4           4.6          3.1           1.5          0.2  Iris-setosa
4   5           5.0          3.6           1.4          0.2  Iris-setosa

In [45]: #dataset = dataset.drop('Id',axis=1)
print(dataset.shape)
dataset.head()

Out[45]:
   id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm Species
0   1           5.1          3.5           1.4          0.2  Iris-setosa
1   2           4.9          3.0           1.4          0.2  Iris-setosa
2   3           4.7          3.2           1.3          0.2  Iris-setosa
3   4           4.6          3.1           1.5          0.2  Iris-setosa
4   5           5.0          3.6           1.4          0.2  Iris-setosa

```

In the Run & Save pane click on a run...

This opens a full reproduction of the entire notebook as it existed at that point in history

The screenshot shows a Jupyter Notebook interface. On the left, there's a sidebar with tabs for 'Files', 'Running', 'Commands', 'Cell Tools', 'Tabs', and 'History'. The 'History' tab is active, showing a list of runs under 'RUNS TODAY AUGUST 22 2018'. One run at 9:22pm is selected and highlighted with a blue background. The main workspace shows two code cells and their outputs. The first cell (In [48]) contains code to read an Iris dataset and print its head. The second cell (In [49]) contains code to drop the 'Id' column and print the dataset shape. Both cells have their outputs displayed below them, showing the first few rows of the Iris dataset.

Now your turn!



this is a prototype, so please don't worry if a bug or break occurs when you're using the tool



feel free to refer back to this overview document at any time

SCAVENGER HUNT TASKS

The following task sets come directly from the survey we ran at Jupytercon 2017, on what people want to retrieve from the past. So these, while challenging, are trying to test the tool's usability on real-life tasks:

- tasks F and L are visual finding tasks such as looking for plots
- tasks D, G, M, I, O, E are coordination tasks which are the hardest, and involve cross-referencing history information about 2 different artifacts eg. what was X when Y was equal to 2?
- tasks H, J, N are code finding tasks which are simpler, involve finding the correct version of just 1 code artifact

Video of best path for each task: <https://www.youtube.com/watch?v=sZpJNh8qlCU>

Intended solution path and features included as arrow bullet points under each question:

A. Find the first version of the notebook

→ scroll to the bottom of the main run map

pane

B. How many cells have been deleted from the

notebook during its history?

→ open up filters and click the cells deleted filter

C. How many runs did the author leave a

comment on?

→ click on the comment filter in the top right

D. Find the code in which the author explored

compare mean absolute error with differing values of max_leaf_nodes

→ search box search for max_leaf_nodes

- E. In RandomForestRegressor(random_state=2),
how many different values of random_state has
the author tried?
→ search box search for
randomForestRegressor and then open up the
inspector view from the results
→ OR click the cell in the current notebook that
sets random forest regressor to open it up in
the inspector view

- F. Find a notebook version that generated a plot
that looks exactly like this:



→ search box search for heatmap

- G. What was the lowest mean_absolute_error
achieved when the author used a
RandomForestRegressor?
→ search box search for
randomForestRegressor and then open up the
inspector view from the results
→ OR click the cell in the current notebook that
sets random forest regressor to open it up in
the inspector view

→ Then, once the versions for random forest regressor have been found, open up the results of that cell in the inspector and look at the values of mean_absolute_error that occurred in the time frame since random forest regressor was first used

H. At what time did the author last use a DecisionTreeRegressor?

→ search box search for DecisionTreeRegressor and then open up the inspector view from the results
→ OR click the cell in the current notebook that sets random forest regressor (and imports DecisionTreeRegressor) to open it up in the inspector view
→ once in the inspector view scroll to the date when decision tree regressor was last used

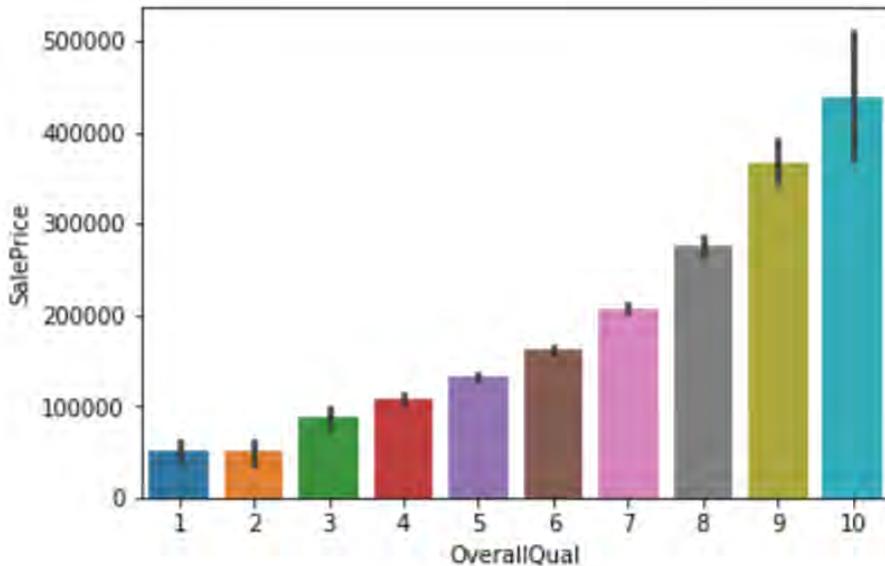
I. What was home_features equal to when the mean_absolute_error(val_y, val_predictions) was equal to below 20,000?

→ find mean_absolute_error and open it up in the inspector. Find the 1 value that's below 20,000 and click ‘these runs’ to see the runs it was used in. From there, open up the ghost book for those runs and scroll (or ctrl-f) to where home_features was set in that notebook

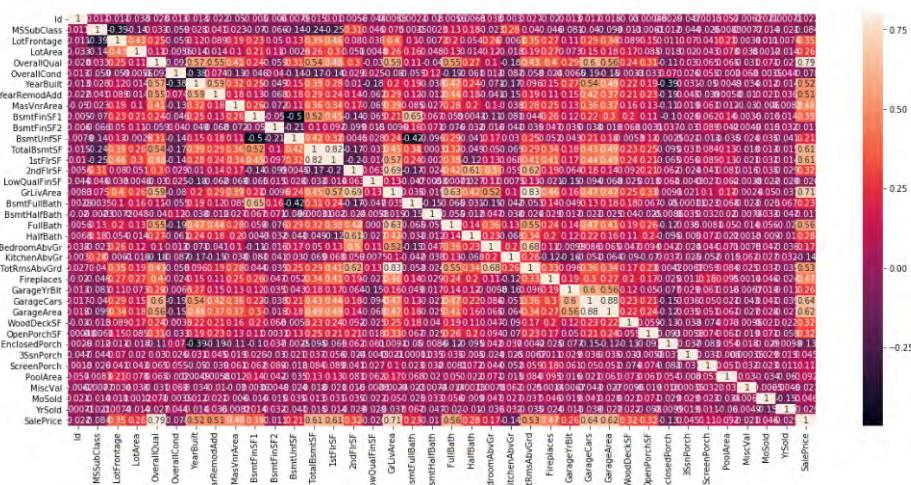
J. Find the code the author used to check for duplicate houses.

→ history search box search for the term “duplicate”. Open the resulting code into a ghost book

- K. Find a notebook version that generated a plot that looks exactly like this:
 → THROWN OUT



- L. Find a notebook version that generated a plot that looks exactly like this:



→ history search “heat map”

M. When “1stFlrSF” was not included in

home_features, did that lower the value of
mean_absolute_error(val_y, val_predictions)?
→ use inspector to look at the history of home
features in the current notebook. Find the 1
version in which 1stFlrSF was not included
→ click “these runs” and open up each ghost
book to check the value of
mean_absolute_error
→ OR note the version time in home_features
for the version that did not include 1stFlrSF, and
using the inspector to check the history of the
cell that sets mean_absolute_error, check what
it’s value was at that timestamp

N. How many different data files eg. “data.csv” has
this notebook been run on?

→ use inspector to check history cell that sets
data file
→ OR use history search to search for “.csv”
and check what .csv files have been used

O. When the author used DecisionTreeRegressor
instead of a RandomForestRegressor, what
parameters did they try for the decision tree?

→ search box search for
DecisionTreeRegressor and then open up the
inspector view from the results
→ OR click the cell in the current notebook that
sets random forest regressor (and imports
DecisionTreeRegressor) to open it up in the
inspector view

PARTICIPANT TASK ASSIGNMENT

We used a Latin Square assignment for participants and tasks. However, since participants had different amounts of time available, not all participants completed the same number of tasks.

Table of tasks completed by each participant ID number in green. Task letters in grey were not completed.

1	A	O	B	N	C	M
2	B	A	C	O	D	N
3	C	B	D	A	E	O
4	D	C	E	B	F	A
5	E	D	F	C	G	B
6	F	E	G	D	H	C
7	G	F	H	E	I	D
8	H	G	I	F	J	E
9	I	H	J	G	K	F
10	J	I	M	H	L	G
11	N	J	L	I	M	H
12	L	O	M	J	N	I
13	M	L	N	A	O	J
14	N	M	O	L	A	B
15	O	N	A	M	B	L
16	A	O	B	N	C	M

Table of success/fail on tasks by participant ID number. Number 1 indicates success while number 0 indicates failure on a given task. Task letters crossed out in pink indicate that task was discarded.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	1	0										1	1	0
2	1	1	1	1	1								1	1	
3	0	1	1	0	0									0	
4	1	1	0	1	1	1									
5		1	1	1	1	1	1	0							
6			1	1	1	0	1								
7				1	1	1	0		1						
8					1	1	1			1	1				

9					1	0		1	1					
10								1	1					
12								1	1	1	0	1	1	
13	0							0		0	0	1	1	
14	1	1							1	1	1	0		
15	1	0							1	1	1	0		
16	1	1	0							1	1	0		

Appendix E: Classroom Deployment Pilot Materials

This study is described in Chapter 12

STUDY SETUP

This pilot study for Verdant was conducted alongside a homework assignment in an AI related course I co-taught. This homework assignment happened to be a programming assignment in Jupyter notebooks, so it was a good opportunity to test out Verdant in real life. Students were required to install Verdant, but not required to interact with it. Students could opt-in to donating their research data logs from Verdant if they chose to consent to our research study. We primarily used data from this pilot to refine and test Verdant's designs using the history logs generated by students.

Protocol

First, students will be introduced to the assignment by the non-researcher instructor and read a verbal script describing that research will be conducted on this assignment. Students will be instructed to install the study tool for this assignment, and be given multiple outside-of-class help sessions at the beginning of the assignment where the researchers will help students install the tool if needed and answer any questions about the study. Students will have the opportunity at the end of the assignment to opt-in to submit their data to the research study. A student who opts out can freely use the tool or not without impacting their grade, and their data will not be kept for research.

During the assignment, the tool in their editor will be passively logging just the student's code experimentation in the code editor and their tool use. The student may freely consult the tool or not as they choose during the assignment. During the assignment, a Piazza thread will be available for students to ask any questions about the study or report any issues with the tool. This thread, as well as email, will be closely monitored by the research team to promptly answer any questions or concerns.

At the end of the assignment, students will be asked to fill out a survey about the assignment, which will ask about the assignment difficulty, experience with the tool, and basic information about their prior programming experience. Students will also be given a consent form at this point to opt-in to the study. Students will submit their log data file along with their code to finish the assignment. If a student has opted out at any point, their log, survey, or grade data will not be given to the research team.

Study Intro Verbal Script

For this assignment, Mary Beth is conducting research on how students explore ideas in code during data or machine learning tasks. We will be providing a new experimental support tool to use with your Jupyter Notebooks for this assignment. Our study involves you completing the assignment as usual, and using the support tool we'll be providing for your experimentation. You are free to use the support tool however you like. Since this tool is new, we will be asked about your experience with this assignment and the tool at the end of the assignment, to improve it for future class use. You will not be asked to do anything above and beyond the normal activities that are part of the course.

For the research study, at the end of the assignment Mary Beth will collect the logs of your code experiments in the Jupyter notebook for this assignment, as recorded by the tool. She will also collect your survey responses and final grade on the assignment, so that we can analyze how well the tool helped or not. Please note we will ask your permission before using any of your data for the study. At the time you submit your homework, you will be given a consent form where it is your choice to submit your assignment data to this study or not. You are free to not participate in this research and your participation will not impact your grade in this course or the assignment. There is no compensation for participation in the study. Finally, if you are under age 18, even if you wish to participate we cannot legally use any of your data for this research.

The analysis is to better understand how to better help students and understand how students work when dealing with machine learning or data programming tasks. For the analysis, the data and course name will be anonymized, and if reported, only reported in aggregate so that it does not tie back or reflect on you as a student.

If you have any questions please contact <other professor> or Mary Beth (contact options in the syllabus). There will also be a place to ask questions on Piazza.

Homework Assignment

Human-AI Interaction, Fall 2019

Assignment #5

The goal of this assignment is to give you hands-on experience with various techniques for generating images using computer vision + machine learning (CVML) techniques... with dogs.



A husky, poodle, and husky-poodle fusion mix generated by BigGan, Assignment #5

Part A: tool setup (15pts)

This assignment requires quite a bit of tool installation, so we will compensate you for your time and effort with 15 points just for getting everything running. Remember: Piazza and Office hours are your friend if you hit installation errors, don't suffer alone!

JupyterLab

Since you'll be using a mix of Jupyter notebooks, text files, and python files, we ask you use JupyterLab, which is kinda like a more built up developer environment that can handle more than just notebooks.

- If you have [Anaconda](#) installed, you already have JupyterLab installed. Open the Anaconda Connect interface and click JupyterLab to start it.
- If you already have JupyterLab, run `jupyter lab --version`. If your version is less than 1.2.2, update it.
- Otherwise, install JupyterLab fresh following [these instructions](#). You will need either the `pip` or `conda` (comes with anaconda or miniconda) Python package managers to do this.

Verdant Log

VerdantLog is an extension for JupyterLab to track the history of your code and output. This is handy since you'll be generating a bunch of images using different parameters.

1. Install NodeJS [using these instructions](#). You can check if your machine already has node by running `node --version`. If possible, update to the latest node version.
2. Install the extension by running `jupyter labextension install verdant-log`. This should take a minute. If it fails the first time, just try the command again.
3. Refresh the JupyterLab app if you had it open. If all works, you should see a log icon  in the leftmost sidebar of the JupyterLab app. Go to a notebook file `HW5.ipynb` and click the log icon to start up the history app for that notebook. In your computer's directory, you'll see the history app creates two files: `HW5.ipyhistory` and `HW5.ipylog`. These are where the history of your notebook is stored.

**** **warning:** We've noticed after generating > 40 images or so, the Verdant Log app can bog down JupyterLab so that it runs super slowly. If that happens to you, go into your directory and dump your `HW5.ipyhistory` and `HW5.ipylog` files into the "Old Logs" folder **but do not delete them** (for grading purposes). Then refresh the JupyterLab app in your browser. This will reset the Verdant Log history starts recording only new history from that point on, and will fix the slowdown. If it's not fixed right away, shut down and restart the JupyterLab app completely. If that fails to fix things (due to dumb memory caching on your machine) restart your computer (sorry!).

Python 3 + TensorFlow 1.15

The models in this homework need Python 3 and TensorFlow 1.15 (*not TensorFlow 2*) to run.

- To check that you have python 3, try `python --version` This should print out something that starts with 3.
- When you open JupyterLab, check that the kernel in the top right corner is set to Python 3:
 If not, click it and change it to Python 3.
- TensorFlow 1.15 can be installed with [these instructions](#), but essentially you should be able to just run `pip install tensorflow==1.15`

Part B: Dog Generation (75 pts)

Finally ready to go! Open up the Assignment 5 folder and then `HW5.ipynb` in JupyterLab and follow the instructions in the notebook to generate some seriously cute (and weird) fake pups.

Turn it in

1. Zip up your entire Assignment 5 folder and submit it to Canvas.
2. Read and fill out the Research Consent Form so we can record whether or not you wish to have your homework data included in a research study. Saying yes or no does not impact your grade whatsoever, and not consenting does not prevent you from completing the assignment; but we need a response from everyone.

Extra credit: Feedback (5 pts)

How well did this assignment go? Fill out the Feedback Survey to get extra credit and help us improve for next year.

Post-Assignment Feedback Survey

The following Survey will be given to students as the final step of their assignment, in a web form format. Participation is optional for extra credit.

Your Name: _____

How much programming experience did you have before you started this class?

- Less than 6 months
- 6 months to 1 year
- 2 - 3 years
- 4 - 5 years
- 6 - 9 years
- 10+ years

How much Python experience did you before you started this class?

- Less than 6 months
- 6 months to 1 year
- 2 - 3 years
- 4 - 5 years
- 6 - 9 years
- 10+ years

How much experience did you have with programming for data science or machine learning tasks before you started this class?

- Less than 6 months
- 6 months to 1 year
- 2 - 3 years
- 4 - 5 years
- 6 - 9 years
- 10+ years

In the following questions, we are looking for feedback on the assignment itself. For each task, please rank how difficult the task was:

(This will be a table of choices, where the rows are each task on the assignment, and the columns are a Likert scale ranging from very easy to very difficult)

In the following questions we are looking for feedback on your use of Verdant:

(The following questions are an adapted System Usability Scale (SUS), with non-SUS questions bolded. These will have a Likert scale ranging from Strongly Disagree to Strongly Agree)

I think that I would like to use this system frequently.

I found the system unnecessarily complex.

I thought the system was easy to use.

I found the system useful for this assignment

I found the various functions in this system were well integrated.

I thought there was too much inconsistency in this system.

I would imagine that most people would learn to use this system very quickly.

I found the system very cumbersome to use.

I felt encouraged to explore more code options by using this system

I found prior versions of my code in the system easily

The following question is an open free text response

Finally, please let us know any feedback of how the system should be improved for future students. Thank you for participating in the study!

Appendix F: The Verdant Study Materials

This study is described in Chapter 13 & 14

STUDY PROTOCOL

A detailed description of the study design is given in Chapter 13. Here are all the accompanying materials to execute the study.

Preflight checklist

1. Send the participant a zoom link

Procedure checklist Session #1

1. In this study we are researching potential roles for experiment history in a data science workflow. We've built a tool that records a lot of log history information while you program. And we're going to see how that information may or may not introduce value to your workflow.
2. Verify payment details and time commitment
3. Verify permission to record the session and **START RECORDING**
4. Tutorial
 - a. Introduce *think aloud* instructions
 - b. Ask them to do all of the questions with the black hidden-answer boxes
 - c. <https://marybethkery.com/Verdant/tutorial/tutorial.html>
5. Give link to server: <link>
6. End tutorial
 - a. From what you've seen of this kind of history functionality, can you see yourself finding this useful in your own workflow?
7. Introduce the tasks
 - a. Introduce that they can use the internet to help with syntax and such just don't go to kaggle
 - b. <Kaggle tasks>
 - c. Get them set up with Verdant, Jupyterlab

Ending procedure

1. How did you find the experience of having Verdant running to the side while you worked during this session?

2. Schedule session 2

Procedure checklist Session #2

1. In this study we are researching potential roles for experiment history in a data science workflow. In the last session we had you do data science coding with our history tool logging what you did. In this session we're going to ask you to use that history to explain aspects of what you did in the first session. We are interested in the usability of our tool as well as how the presence of history data might affect your explanations.
2. Verify permission to record the session and **START RECORDING**
3. Open back up notebook from last time
4. Tutorial from last time
 - a. <https://marybethkery.com/Verdant/tutorial/tutorial.html>
5. Open back up client prompt from last time

Narrative Overview

1. In the last session, how did you approach the client's goals? I'd like you to walk me through what kinds of things you tried and what you found in Session 1.

Tasks: History Finding Questions

1. Explain in this section there will be 10 questions. Each has two parts and the participant will be using the Verdant tool features to answer the questions. In the first 3 they will be instructed with a specific feature of Verdant to start their search on, and after that training phase they can use any feature they choose in Verdant.
2. This is a tool *prototype* so you could encounter some bugs or issues
3. Instruct the participant to **THINK ALOUD** and confirm that they understand
4. Give questions 1 at a time, and 1 piece at a time into Zoom chat

Interview: Explaining History Access Questions

1. *In answering those tasks you just answered, did you personally think looking at the history artifacts was more helpful or more unnecessary?*
2. *Compared to real questions you might be asked about in a meeting, did you find the questions you just answer more realistic or more contrived?*

3. What do you see as the use of returning to earlier parts of your experimentation? In what circumstances do you believe the history of your analysis work is useful? In what circumstances do you believe that the history of your analysis work is unnecessary.
4. Are there any parts of the data analysis or modeling process that you think would be benefited by better engagement with the history of what you've experimented with so far and why?

Interview: Verdant Usability

5. *What do you think about the overall usability of the Verdant tool? What do you like about it? What do you not like about it?*
6. Are there features that you wish it included? Are there any parts of the features you found confusing?
7. How easy do you think it is to get back to a prior part of your experiment history using Verdant?
8. *How easy do you think it is to get back to a prior part of your work **without** Verdant? What barriers might make it harder to get back to a prior part of your work?*

Tutorial

The screenshot shows the GitHub page for the Verdant project. The title "VERDANT" is prominently displayed. A description follows: "A version control tool for JupyterLab that automatically records the history of your experimentation while you work." Installation instructions are provided: "To install: `jupyter labextension install verdant-history` or search `verdant-history` under the extension manager tab in JupyterLab." Navigation links "Overview" and "Getting Started Tutorial" are visible.

Welcome! The goal of this getting started tutorial is to give you some hands-on practice understanding the edit history of a Jupyter notebook using Verdant.

Setup & Materials

For this tutorial you'll need Verdant installed in JupyterLab. We'll use a notebook `houses_kc.ipynb` and its Verdant history file `houses_kc.ipynbhistory`. Verdant stores output images, like plots or charts, in a separate folder `houses_kc_output`. All of these materials can be downloaded as a zip file `tutorial_notebook.zip` from the [GitHub repository tutorial folder](#).

Next unzip the folder on your own machine and start jupyter lab with `jupyter lab`. Open up the notebook in: `tutorial_notebook/houses_king_county/houses_kc.ipynb`.

If everything has loaded OK, you should see **Verdant's tree log icon** at the left. Click the log to open up Verdant. You should see an activity pane like below.

The screenshot shows a JupyterLab interface with the Verdant extension active. The left sidebar displays an "Activity" pane with a tree log showing entries for "YESTERDAY JANUARY 25 2021" at 4:50pm, specifically listing "v109 - v92". The main notebook area shows a code cell starting with `[61]: # import our basic libraries` and `import pandas as pd`.

The screenshot shows a Jupyter Notebook interface. On the left, there is an activity pane displaying a list of notebook versions with timestamps and commit IDs. A mouse cursor is hovering over the first item, v91, which is circled in red. To the right of the activity pane is the main notebook content area. The code cell contains imports for matplotlib, seaborn, sklearn, and numpy, followed by a title 'Predicting housing prices in King County' and a brief description of King County's geography. Below the code cell is a text block with a note about triple-clicking the black bar to reveal answers. The main content area also includes a large black rectangular redaction box.

In `houses_kc.ipynb`, the author has been at work predicting housing prices in the Seattle area of the United States. In this tutorial, we'll use history to uncover different details of the analysis and the author's modeling choices.

In this tutorial, triple-click the black bar with your mouse to reveal the answers to a question.

1. Get an overview of what has been done so far

The dataset in this analysis has a fair number of features to describe each property. However, the author of `houses_kc.ipynb` seems to have picked just a single feature `sqft_living` for their first model:

Baseline model

For a baseline regression model, we'll just take our strongest single feature (`sqft_living`) and create a single feature regressor to see how that does.

Did the author explore any of the other features? One good way to get an overview is to take a peek at how this notebook has evolved over time. Using the **activity pane** (shown below) scroll down to the earliest version of the notebook.

The activity pane shows a list of notebook versions. The first item is expanded, showing the timestamp 'MONDAY JANUARY 25 2021', the commit ID 'v109 - v92', and a scroll bar on the right.

Activity	Artifacts
MONDAY JANUARY 25 2021 (109)	
4:50pm... v109 - v92	
4:46pm v91	
4:45pm v90	



Question: Looking at the earliest version, what date and time did the author first start working on this notebook?

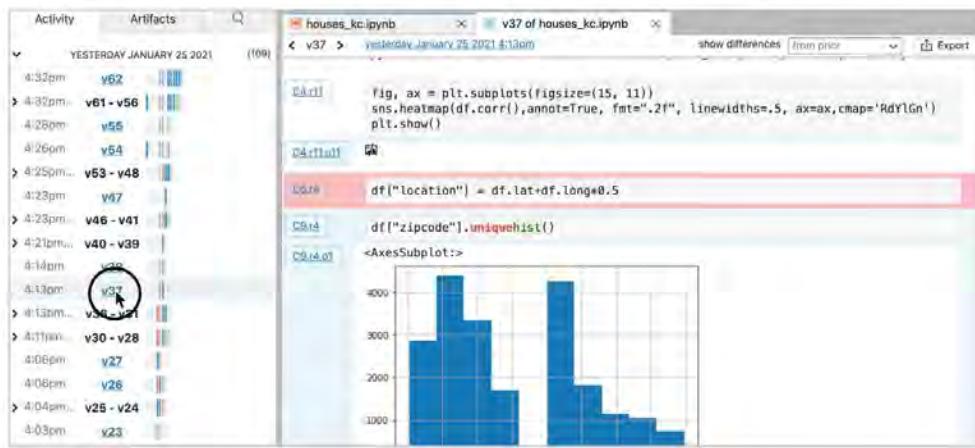


The **notebook minimap** with the colorful ticks next to each version row shows us which cells changed in a given notebook version. Green indicates a cell added, red a cell deleted, blue a cell changed, and so on.

Question: Use your mouse to hover over the vertical line tick in the minimap for **v39-40**. Which cell was changed in v39-v40 and how?

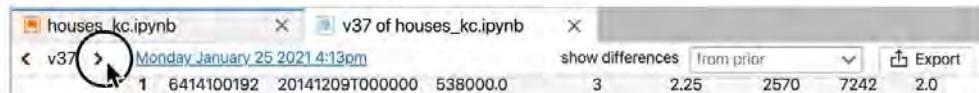


Now let's get back to figuring out what the author tried in this analysis. Try clicking on a version **v37** (shown below) to open the full notebook at that version as a **ghost notebook**:



The diff notation in the **ghost notebook** highlights what content was added (green), deleted (red), or edited (blue) in this version. Next, try using the arrows at the top of the ghost book to

move ahead in time a couple of versions, and watch how the notebook changes:



Question: Glance through versions v37 to v50 of the notebook. Other than sqft_living, what are 2 other features the author plotted compared to price?



Hopefully you've spotted a few charts in these versions, however it's a bit tedious to skim one at a time though versions. Next let's learn some strategies to more directly find specific things from history!

2. Track model improvements

For this notebook, it would be helpful to know what kind of model performance the author has gotten so far for predicting house prices. What kind of models have they tried and what were their results?

There are a few ways in Verdant to find specific history. Head over to the Artifacts tab.

The Artifacts tab has two views: a **table view** summarizing overall history of your current notebook and a **detailed history view** that will show you all history for one specific artifact (a cell or an output):

Activity	Artifacts
houses_kc.ipynb v133 by artifact revisions: C1 6 # import our basic M1 4 Predicting housing p C2 3 df = pd.read_csv(.. out id	NOTEBOOK > CODE CELL 13 C13 r16 created in Notebook v127 1:25pm Jan 29, 2021 <pre>x = np.array(df['grade'] * sqft_living) y = np.array(df['price']) # what's the relationship?</pre> C13 r15 created in Notebook v125 1:25pm Jan 29, 2021 <pre>x = np.array(df['sqft_living'] * grade) y = np.array(df['price']) # what's the relationship?</pre> r14

Click Notebook in the nav bar to go back to the table view

First, let's check out the table view:

Activity			Artifacts		Q
houses_kc.ipynb v129 by artifact revisions:					
cell	revision	preview			
C1	6	# import our basi...			
M1	4	Predicting housing p King County in the USA			
C2	3	df = pd.read_csv(...			
	out		id		
			0	7129300520	20141
			1	6414100192	20141
			2	5631500400	20150
			3	2487200875	20141



Version Inspector

For each cell & output in your current notebook, this table lists how many versions exist of that cell. Cells are abbreviated **C** for code and **M** for markdown. Each cell has a number that identifies it, like **C2**. This id is unique, permanent, and never changes, so that you can always

recover a specific cell even after it's been deleted.

Question: How many versions exist for C4 (Code Cell 4)?



Scroll the table down to the output of code cell 14 (C14). In the preview of this output, we can see that this cell of the notebook prints out model metrics.

```
C14    7    from sklearn.metrics imp...
      out
      4
      →
Coefficients:
[273.80181792]
Mean squared error: 721691
Coefficient of determinati
```

Click on the output to see all of its versions:

Notebook will take you back to the table view

Activity Artifacts

NOTEBOOK > CODE CELL 14 > OUTPUT

C14.r7.04 created in Notebook v128
1:29pm Jan 29, 2021

code C14.r7

Coefficients:
[273.80181792]
Mean squared error: 72169122334.17
Coefficient of determination: 0.50

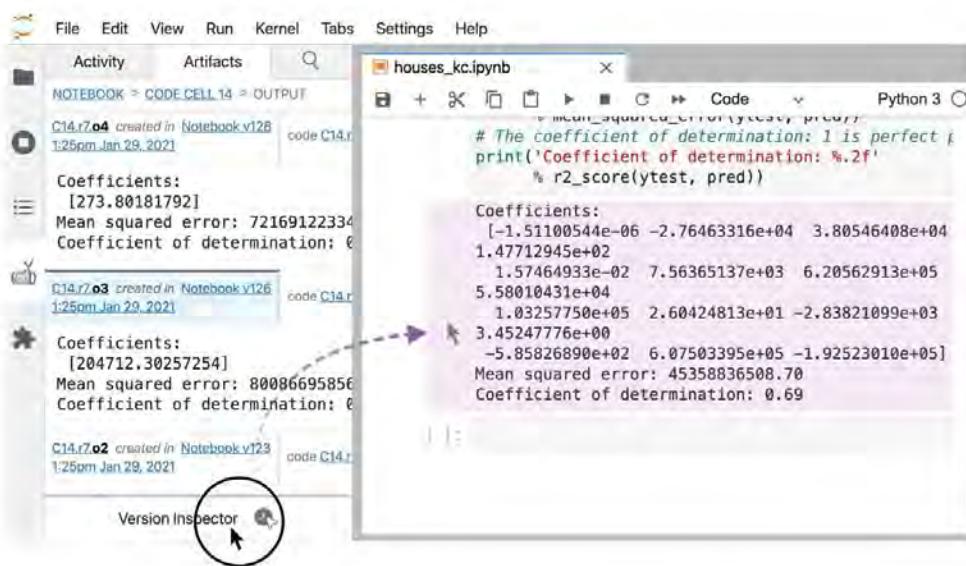
scroll to see all versions of this artifact (Output of C14)

Welcome to the **detail view**! The detail view will show you every version that has ever been created of a specific thing. While in this view, clicking on a link like Notebook v128 will open up the **ghost notebook** to help you see a specific version in context of the full notebook version.

Question: Using the detail view for the **output of code cell 14**, what was the lowest value of **Coefficient of determination** the author got?



Now, taking a look at the current Jupyter notebook, we can see that the author actually has two different models in their notebook. Next let's take a look at the history of metrics for the second model.



A different way of finding history in Verdant is to use the **version inspector**.

Click on the **Version Inspector** button to activate the inspector. Then click on the final output in the current notebook to see its history. With the inspector, you can get the history of **anything** from the current notebook just by clicking on it. After using the inspector to click on the last output of the notebook, you should now be seeing metrics of the output of code cell 22:

Activity Artifacts

NOTEBOOK > CODE CELL 22 > OUTPUT

C22.r2.o3 created in Notebook v129 1:26pm Jan 29, 2021 code C22.r2 >

Coefficients:

$[-1.51100544e-06, -7.76463316e+04]$
 $[5.7484933e-07, 7.5655137e-08]$

Question: Using the detail view for the **output of code cell 22**, what was the highest value of **Coefficient of determination** the author got?

Finally, sometimes you'll want to retrieve the history of a cell or output that was **deleted** i.e. it no longer exists in the current Jupyter notebook. To find history of anything by keyword, *including* the history of previously deleted cells, use the **history search** tab of Verdant:

The screenshot shows the Verdant interface with a search bar containing the text "Coefficient of determination". Below the search bar, there are three items listed under the "Artifacts" tab:

- > appears in (3) code cells
- > appears in (4) markdown cells
- > appears in (8) outputs

3. Finding images, charts & visualizations

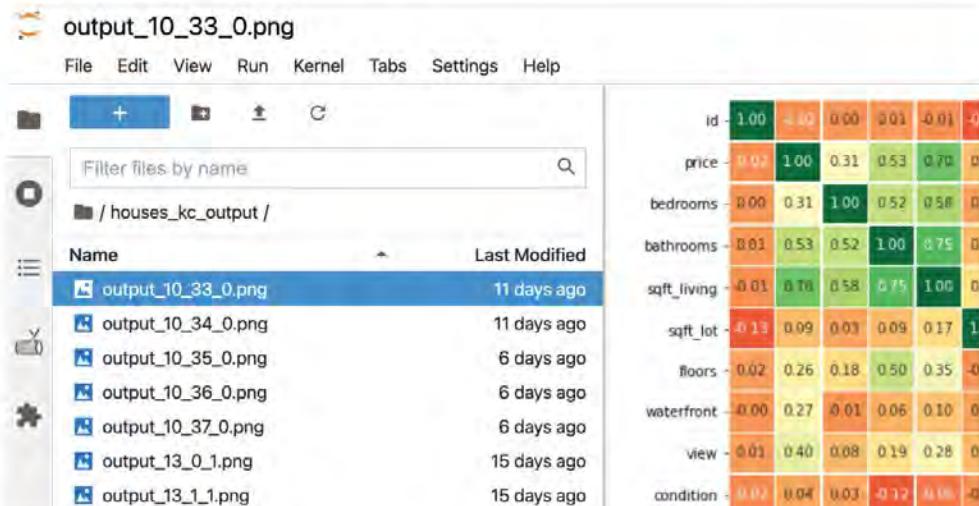
A few ending words on how to find charts and images in history. The **history search** has a few special keywords, such that if you search for "plot" or "image" or "chart", the search will return *all* images a notebook has ever produced:

Activity Artifacts



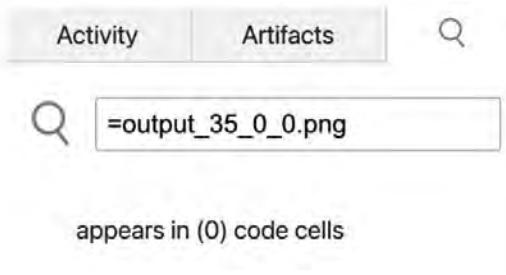
Verdant actually stores all images a notebook produces in an auto-generated folder titled {name of your notebook}_output such as `houses_kc_output`. Open that folder in your normal computer file browser to see all the images. If you decide to delete images stored in this folder they will just show up with a missing image symbol in Verdant:

[C4.r11.o16 from Notebook v56](#)



Meanwhile if you want to *reproduce* an image you find in the output folder, simply copy the name of the file (shown below in JupyterLab's file browser):

Be sure to type `=` before the name to let the history search know you're searching for an exact artifact by name:



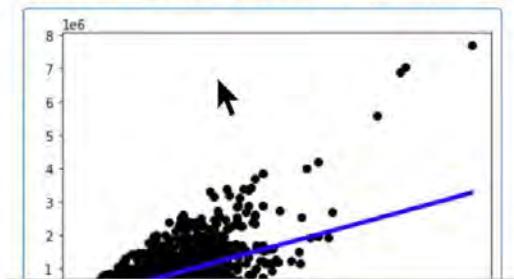
Activity Artifacts

appears in (0) code cells

appears in (0) markdown cells

▼ appears in (1) output

[Code Cell 17 Output](#) C17.r3.o1 from [Notebook v73](#)



Question: One final question to put it all together. How many versions of code cell 17's output exist?

[REDACTED]

You did it. Happy coding! If you encounter any issues with Verdant, we always appreciate [feedback](#) or a issue report in the [repo](#). Many thanks from our team at Carnegie Mellon University. <3

[Verdant](#) is licensed under the MIT License and is maintained by [Mary_Beth_Kery](#) at Carnegie Mellon University <3

Programming Task

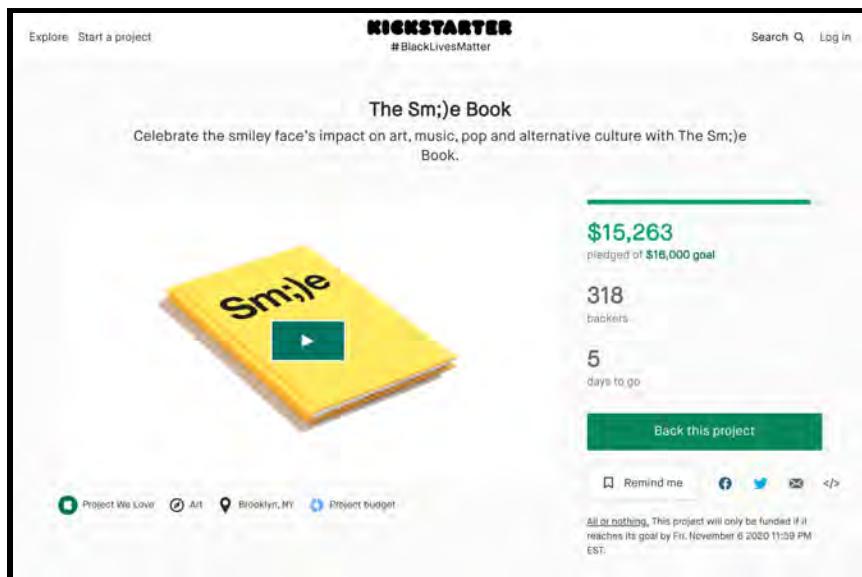
Verdant Usage Study: Session 1 Tasks

Goal Overview

Kickstarter is an online platform where people contribute money to *crowdfund* a project such as a book, game, or music. With this startup funding from the crowd, the project team is able to produce their idea as a real-world product.

You are given a dataset of over 300,000 Kickstarter projects. As Kickstarter explains it:

"Every project creator sets their project's funding goal and deadline. If people like the project, they can pledge money to make it happen. If the project succeeds in reaching its funding goal, all backers' credit cards are charged when time expires. Funding on Kickstarter is all-or-nothing. If the project falls short of its funding goal, no one is charged."



Example of a typical Kickstarter project

Client: You are being hired by a business school professor who wants to teach students about crowdfunding in their “Internet Entrepreneurship” course where students will create their own Kickstarter startup pitches for a class project.

Goals:

1. Create an exploratory data analysis that can inform your client about what makes a *successful* Kickstarter project. A successful crowdfunding project reaches its complete financial goal.

2. Your client would also like you to communicate any interesting patterns about what might make a Kickstarter project *fail*.
3. Your client wants to know if you can build a machine learning model such that a student can input data about *their* project idea and your model will output how successful that project is likely to be. However, they would also accept if Kickstarter projects are not that easy to predict. The professor wants you to tell them if such a model is actually impractical.

Packages: To install a missing package, type and run **%pip install <package>** in a cell in your active notebook

Deliverables: In Session 2 (roughly a week from this study session 1), you will be asked to present & discuss your work for this client in a Jupyter notebook format.

Note that this study is not evaluating your level of coding, statistics, or machine learning skills. Please work as you normally would.

Note the dataset for this study comes from [kaggle.com](https://www.kaggle.com). You are free to use the internet to help you code, just **do not go to kaggle.com** or any place you believe will directly give you the “answers” for the client’s requests about this dataset

Scavenger Hunt Questions

Participant	Order	Question Type	Question Text 1	Question Text 2	Required Start Feature	Answered?	Answered using history?
P01	1	Modeling	Find: Did you test different alpha values for the lasso for each new feature you added?	Explain: why or why not?	Inspector	TRUE	TRUE
P01	2	Visualization	Find: Go back to the plot where you had x = usd goal bin, y = project	Explain: What might a plot like this potentially tell you that a table of	Artifact Summary	TRUE	TRUE

			count, and hue = main category.	the mean usd goal bin might not?			
P01	3	Feature creation/selection	Find: Did you use or look at the country feature at any point?	Explain: Do you hypothesize the country feature would be helpful or not as an addition to the model?	Search	TRUE	TRUE
P01	4	Feature creation/selection	Find: when you were first creating the binning for usd_goal_real	Explain: What did you look at or how did you decide on your binning function for usd_goal_real?		TRUE	TRUE
P01	5	Modeling	Find: at some point, you had the output from the Lasso print for each feature true/false whether its coefficient == 0	Explain: what kind of information did this true/false tell you? Why did you later choose to filter out some features from being displayed in the model output based on this information?		TRUE	TRUE
P01	6	Data Cleaning & Filtering	Find: where you originally used project ID as the index for df_finished	Explain: why did you initially choose to use the project ID as the index, and then why did you later switch to a different indexing?		TRUE	TRUE
P01	7	Storytelling & Notebook Organization	Find: some point in time where your notebook contained a to-do note	Explain: how is this to-do note similar or different from the kind of note taking you might normally do in your data science coding work?		TRUE	TRUE
P01	8	Feature creation/selection	Find: an earlier version of df_finished_dummies, where you included year	Explain: What was the effect on the predictive accuracy of your model when you		TRUE	TRUE

			as a dummy categorical variable	counted year as a categorical variable versus a numerical variable?			
P01	9	Feature creation/selection	Find: earlier when you used usd_goal_real as a plain numerical feature rather than a binned feature	Explain: comparing binning and not binning usd_goal_real, what was the effect of binning on the model performance?		TRUE	TRUE
P01	10	Visualization	Find: got to when the plot with x = num samples, y = main category, hue = usd goal bin was a table before it was a plot	Explain: several of your plots seem to develop this way, first as tables that you then turn into plots. Is this a common workflow for you and do you use the plot or the table or both when you need to debug a plot's design?		TRUE	TRUE
P02	1	Modeling	Find: the last notebook in which you used a decision tree classifier as your model.	Explain: What prompted your decision to switch from the decision tree classifier to random forest?	Inspector	TRUE	TRUE
P02	2	Data Cleaning & Filtering	Find: In an earlier version, due to a spelling typo, dfFiltered contains just kickstarter projects that failed, and not those that succeeded. Find the heatmap for dfFiltered correlations when dfFiltered only contained failed projects, and later when it contained both failed + successful projects.	Explain: how does the correlations for the feature pledged change between the heatmap showing failed projects only versus the heatmap showing both failed + successful projects?	Artifact Summary	TRUE	TRUE

P02	3	Data Cleaning & Filtering	Find: When the variable freqTable_f was first created	Explain: What was your purpose for freqTable_f as a print statement for your model output?	Search	TRUE	TRUE
P02	4	Visualization	Find: In the scatter plot for x = DaysOpen and y = goal, you initially had a few outliers skewing the whole plot. Find the scatter plot back when it showed the outlier points	Explain: Approximately how many days open were the outlier kickstarter projects. Can you hypothesize why kickstarter project creators would choose such an high value?		TRUE	TRUE
P02	5	Tables & Summary Stats	Find: Go back to when the categPiv pivot table contained raw counts instead of percentages	Explain: By raw count, which 3 categories had the fewest live project campaigns running when this data was collected? Compare to the historical percentage of that category that is successful. If a student were to launch a kickstarter project in one of those 3 categories, which category would you advise has the best chance of success?		TRUE	TRUE
P02	6	Feature creation/selection	Find: Did you ever use OverGoal as a predictive feature?	Explain: What was your intention in creating df["OverGoal"] and what did you want to use it for?		TRUE	TRUE
P02	7	Feature creation/selection	Find: when StartMonth and Deadline month were included as features in the model	Explain: Why had you chosen to try adding these features to the model? What was the apparent effect of each of these features on		TRUE	TRUE

				the performance metrics of the model?			
P02	8	Modeling	Find: the result you got from adding the feature OneHotCurrencyName to the model	Explain: What was your hypothesis about why this feature might matter for a kickstarter project's success?		TRUE	TRUE
P02	9	Visualization	Find: some point in the notebook's history at which you had a visualization of the decision tree model output	Explain: What kinds of things did you attempt in visualizing the decision tree and what kind of barriers did you run into?		TRUE	TRUE
P02	10	Modeling	Find: when you switched the max depth of the tree from 2 to 5	Explain: What were your motivations in changing the max depth and what kinds of results did you see?		TRUE	TRUE
P03	1	Modeling	Find: Did you do any hyperparameter tuning on either the logistic regression or the random forest?	Explain: Do you feel you reached a place with the models that you wanted to be doing hyperparameter tuning? Why or why not?	Inspector	TRUE	TRUE
P03	2	Modeling	Find: What was the best accuracy score you got for logistic regression?	Explain: Why did you decide to switch away from logistic regression?	Search	TRUE	TRUE
P03	3	Tables & Summary Stats	Find: Go back to when you had a table showing each possible project state (failed, successful, live, canceled, etc.) grouped by the project's currency	Explain: At the time this dataset was collected, which currencies had the most live projects on Kickstarter?	Artifact Summary	TRUE	TRUE

P03	4	Visualization	Find: Earlier when you had the name length plot as a violin plot and when you had it as a bar chart	Explain: You tend to use bar, box, and violin plots a lot in this notebook. Is there something different in what the different chart styles tell you about the name length feature?		TRUE	TRUE
P03	5	Feature creation/selection	Find: You created a days until deadline feature. Did you ever use it in your model?	Explain: Do you think this feature could help predictive accuracy, why or why not?		TRUE	TRUE
P03	6	Storytelling & Notebook Organization	Find: You noted that projects launched on the 1st or 31st of the month may have higher success rates. I see you have a plot comparing success/fail across each date. Did you ever calculate the percentage success rate for each day of the month or are you basing your hypothesis on just the chart?	Explain: Do you have any intuitions about why picking the right day of the month might help a project's success?		TRUE	FALSE
P04	1	Feature creation/selection	Find: Where you compared the values of the duration days feature between success/fail, if you did explore that.	Explain: Did you have any insights about the difference between success and failure in terms of duration of the project campaign?	Search	TRUE	TRUE
P04	2	Visualization	Find: There doesn't appear to be a pairplot in the current notebook. Did you at any point have a pairplot showing?	Explain: What were you hoping to visualize with the pairplot?	Artifact Summary	TRUE	TRUE

P04	3	Feature creation/selection	Find: Go back to when you initially created drop_fields.	Explain: Initially you had just a few features listed, and then added a whole bunch of features. What prompted that and how did you choose what to drop?		TRUE	TRUE
P04	4	Tables & Summary Stats	Find: In your notebook, all the data frame table displays appear to be you transform the data somehow and then use the head() function to show the transform's effect. Did you use tables in other ways, such as showing summary stats for your data frames or group-by tables?	Explain: How much do you typically use table displays as opposed to visualizations to explore datasets in your data science practice?		TRUE	FALSE
P04	5	Storytelling & Notebook Organization	Find: In a markdown note you note that since most of the currency is USD, it might not tell us much about success/failure. At the same time, you use the one hot encoding of the categorical currency feature in the model. Did you make this hypothesis before or after trying currency in the model, i.e. did you test this hypothesis?	Explain: What is your current intuition about how helpful the currency feature is?		FALS E	
P04	6	Modeling	Find: Go back to when initially you had 99% accuracy on your model.	Explain: As you developed the model, over time its accuracy went down, why is that?		TRUE	TRUE

P04	7	Data Cleaning & Filtering	Find: Your notebook contains a validation set. Did you ever run the model using it?	Explain: At what point of your model development process would you have used the validation set?		TRUE	TRUE
P04	8	Feature creation/selection	Find: What kind of performance change did you see in including the smaller categories feature in your model?	Explain: What is your intuition about how helpful the categories feature is?		TRUE	TRUE
P06	1	Feature creation/selection	Find: Did you ever try a different feature set than 'category', 'main_category', 'country', 'currency', goal, delta_t?	Explain: What reasons do you have for including all of the category variables when those overlap with main_category?	Inspector	TRUE	TRUE
P06	2	Tables Summary Stats	Find/Explain: Having no country as Country N,0" seems to be highly predictive of a project failing. Did you ever do anything with Country N,0" or filter those projects out of the dataset? Why or why not?		Search	TRUE	TRUE
P06	3	Modeling	Find: What effect did scaling the variables log_goal and log_delta_t have on the overall model performance?		Artifact Summary	TRUE	TRUE
P06	4	Feature creation/selection	Find: Go back to when variable s was first created	Explain: What was the purpose of s and was it ever used for anything?		TRUE	TRUE
P06	5	Visualization	Find: Go back to the plot of df['usd pledged'] / df['usd_pledged_real']	Explain: What were you trying to find out about df['usd pledged']		TRUE	TRUE

			before you made the plot log scale	/ df['usd_pledged_real'] by playing around with the scale of the plot?			
P06	6	Visualization	Find: the version of the pairplot before log scale was applied	Explain: In the pairplot some comparisons become much more visible when the log scale is added while others are much more visible without the log scale. What can you hypothesize about backers x pledged from the pairplot? Or goal x usd_goal_real?		TRUE	TRUE
P07	1	Feature creation/selection	Find/Explain: What did you try in your analysis with the “state feature”		Inspector	TRUE	FALSE
P07	2	Data Cleaning & Filtering	Find/Explain: There's df2, df2_onehot, and onehot in the notebook, which are all variables that appear to have to do with onehot encoding but don't end up in the model later in the notebook. Please use history to explain what these were used for.		Artifact Summary	TRUE	TRUE
P07	3	Tables Summary Stats	& Did you do any exploration of the “main_category” feature?		Search	TRUE	TRUE
P07	4	Tables Summary Stats	& Find/Explain: During the notebook's history, what was the variable “j” used for?			TRUE	TRUE
P07	5	Visualization	Find: Go back to when the histogram of	Explain: What were you aiming to do in		TRUE	TRUE

			“category” feature was brown-green	changing around the colormap from blue to brown to red?			
P08	1	Modeling	Find: Go back to when you had a scatter plot of <code>x = live_time, y = backers, hue = success/fail</code>	Explain: What, if anything, can you hypothesize about the relationship between <code>live_time</code> and <code>success/fail</code> from this plot?	Inspector	TRUE	TRUE
P08	2	Feature creation/selection	Find: Go back to when Coefficient of determination: 0.95.	Explain: What caused such a high coefficient of determination? Also why 0.95 rather than 1.0?	Artifact Summary	TRUE	TRUE
P08	3	Visualization	Find/Explain: What different features did you try in the model?		Search	TRUE	TRUE
P08	4	Data Cleaning & Filtering	Find/Explain: You have this nice analysis looking at the percentage success of projects by <code>main_category</code> . Did you apply that analysis to “category” or any of the other categorical features?			TRUE	TRUE
P08	5	Modeling	Find/Explain: How did you decide to handle projects with other states than success/failed, such as canceled?			TRUE	TRUE
P09	1	Feature creation/selection	Find/Explain: Which variable combinations did you try looking at the relationships between?		Artifact Summary	TRUE	TRUE

P09	2	Data Cleaning & Filtering	Find/Explain: Can you walk through what you tried for the df where df.country is N,0" data filter		Inspector	TRUE	TRUE
P09	3	Tables Summary Stats	Find: What the variable "countries" was used for in the history of the notebook	Explain: What kind of analysis were you seeking to do with the country feature?	Search	TRUE	TRUE
P09	4	Feature creation/selection	Find/Explain: How did you use the feature cat_code?			TRUE	TRUE
P09	5	Modeling	Find: Back when you had y = pledged x = category what model result did you get?	Explain: What did that tell you about the relationship between pledged and category?		TRUE	TRUE
P10	1	Visualization	Find: Go back to when you had a plot for comics	Explain: Are there any categories of comics that are substantially more successful than others?	Search	TRUE	TRUE
P10	2	Storytelling & Notebook Organization	Find/Explain: Did you have any insight into why Hiphop is such an unpopular kickstarter project category?		Inspector	TRUE	TRUE
P10	3	Tables Summary Stats	Find: Did you look at the distribution of monetary goal for successful projects versus failed projects?	Explain: Do you have any hypothesis about how helpful monetary goal would be as a feature?	Artifact Summary	TRUE	TRUE
P10	4	Visualization	Find: Go back to when you had a plot of the feature duration_in_seconds	Explain: Do you see any trends about duration_in_seconds between successful or failed projects?		TRUE	TRUE

P10	5	Data Cleaning & Filtering	Find: Go back to when you had a dataframe named was_successful	Explain: Before you found the state feature, how were you calculating success/failure?		TRUE	TRUE
P11	1	Visualization	Find: Go back to when you had a plot showing the amount of each main_category project across all states, including states like undefined or canceled	Explain: Do you see any differences in how these “other” states are distributed across different categories, such as some categories having more cancelled projects than others?	Inspector	TRUE	TRUE
P11	2	Data Cleaning & Filtering	Find: When you did a visualization with the country feature	Explain: What is your hypothesis about how helpful a feature country is?	Search	TRUE	TRUE
P11	3	Feature creation/selection	Find: You include category as a predictive feature for the model. Did you do any analysis of the category feature and if so what did that tell you?	Explain: Do you think categories may have additional predictive power than just using the main category?	Artifact Summary	TRUE	TRUE
P11	4	Tables & Summary Stats	Find: Go back to when you had an analysis of projects with the country value N,0"	Explain: You ended up not filtering projects with the country of N,0" out of the analysis? Why did you decide to keep them in?		TRUE	TRUE

Thematic Analysis Qualitative Codes

- 1. Search Feature
- 2. Inspector Feature
- 3. Activity Feature
- 4. Ghost Notebook Feature

- 5. Artifact Table Feature
 - 6. Artifact Detail Feature
 - 7. Navigation troubles
 - 8. Navigation is easy
 - 9. Unsure how Verdant works
 - 10. Verdant usability
 - 11. Information overload
 - 12. Horizontal width of Verdant pane
 - 13. Naming/numbering in Verdant
 - 14. Feature suggestion
 - 15. Using Verdant post-study
 - 16. History in other notebook systems
 - 17. Computation overhead
 - 18. History is helpful
 - 19. History is more important for collaboration/onboarding
 - 20. Verdant-style history is not useful for team work
 - 21. Notebook is hard for collaboration
 - 22. History of plots/output
 - 23. History for model performance
 - 24. Using/checking history while working
 - 25. Using history to debug
 - 26. History changes my notebook practices
 - 27. NOT using history while working
 - 28. Cell-based history requires a certain workflow
 - 29. Data work you don't remember
 - 30. I can just answer this from memory
 - 31. Keep track of deleted content
 - 32. Something is lost
 - 33. Re-code something lost
 - 34. Automatic history keeping
 - 35. Github with notebooks
 - 36. Output history kept in folders locally or in the cloud
 - 37. Code/non-code history live in different places
 - 38. Automatically log model metrics to somewhere
 - 39. Manually creating a logging pipeline
 - 40. Using tools to automate logging pipeline
 - 41. Use external notes doc to keep history
 - 42. Informal versioning: keep multiple copies
 - 43. Organizing notebook to keep history
 - 44. History is unorganized
 - 45. Hard to connect outcomes with version of notebook that generated it
 - 46. Pruning notebook
 - 47. show you my thought process
 - 48. When I don't keep history
 - 49. How often I check history
 - 50. When I get history questions
 - 51. What I use notebooks for
 - 52. Data analysis tasks are realistic/unrealistic
 - 53. How long I would need to do this data analysis
 - 54. Other programming languages
 - 55. When I do exploratory work like this
-

Appendix G: Query Design Exercise Results

This study is described in Chapter 4

Summary Table of Results

Does Verdant address this?	Count	Percentage
A little	29	24%
Fully	37	30%
N/A	12	10%
Not at all	13	11%
Partially	32	26%
Grand Total	123	100%

Full list of Queries

Raw text of question	How would a user answer this in Verdant?	Does Verdant address this?
This one time, I got an output of <x>. What parameters was I using to generate that?	Verdant: from output <x> open ghost book to see code that generated it	Fully
When I found a correlation matrix in which <x> was positively correlated with <y>, 0.41.... and <q> was negatively correlated with <z> -0.03, ...what correlation measure was I using?	Verdant: first find that specific correlation matrix, then find the code that generated it	Fully
Also, how had I operationalized measures and <c> at that point? Is it the same operationalization as I'm using now, or different. I.e., do I have to do this over again?	Verdant: look at ghost book at that time in history to figure out the operationalization	Partially
What results did I get when I tried using analysis/algorithm <a> on <this exact dataset>. (did I even do that already, or am I imagining things?)	Verdant: look for where <a> occurred. Depending on the file and variable name, it may or may not be possible to look up the exact dataset	Partially

<p>How the fuck did I end up with this intermediate value of 0.43 for this one parameter? No seriously. This is the worst. Looks like I hardcoded it in here at some point... so at some point, I must have done some sort of analysis to arrive at this value, and then subsequently plugged it in to my future analyses... and then like, deleted that intermediate analysis? UGH. What assumptions was I making in whatever analysis I did to get that. What algorithm was I even using? Is this even... is this like supposed to be here? Or was it an unrelated analysis, and I plugged in 0.43 from the wrong place. I was kind of sleepless when I was doing this.</p>	<p>Verdant: search history for the value of 0.43 and that should point to where in history it occurred. This may be able to get you back to the analysis but won't tell you what assumptions you were making</p>	<p>Partially</p>
<p>How many different ways did I run [a particular regression] (e.g. what variables left in, left out)?</p>	<p>Verdant: get history of that particular regression, see all combinations it was run with</p>	<p>Fully</p>
<p>When did I run the same thing (e.g. the same regression model), but get different results? [<-- sometimes I would change something about the underlying data, as it was just in CSV format, and then forget I had done that...].</p>	<p>Nope: a regression may output something a little different each time, so it may be hard to tell where the data changed if the data changed outside of the notebook. There might be some evidence of the data changing in the output of the notebook, it depends</p>	<p>A little</p>
<p>When did I make changes to the underlying data, and what is the list of analyses I ran prior to and post making changes?</p>	<p>Again, it depends if they made changes to the underlying data in the notebook or outside of it. If the changes were made in the notebook, this should be partially doable to get a date and time of that change. But even so, Verdant can't collate a nice list of analyses after a certain time. Verdant might give enough hints to</p>	<p>A little</p>

	solve this question, but cannot answer it directly	
What were all the different statistical functions I used, and how many times did I use each one? [-- might use this to remember what kinds of analyses I was doing since I've kind of forgotten, and then might query further based on the results]	No	Not at all
Show me the last regression I ran	Verdant: search for the term "regression" or if you recall the name of the regression. If the regression has a really specific name, it may be harder to find.	Partially
Show me the settings I used for that last analysis	Verdant: first locate that last analysis. Then look at the surrounding code with the ghost notebook	Fully
Show me the last four crosstabs	Verdant: search for "crosstabs". Again, if the crosstabs are named something different or not named the same thing, this may be hard to find	Partially
Show me results from feature extraction on XX date	Verdant: since they have a specific date and specific analysis in mind, this should be pretty easily searchable using the activity and search features	Fully
Revert model to parameters used in trial XX	Verdant: since they have a specific model and iteration in mind, should be easily searchable	Fully
Effects of [insert name of variable X] on [insert name of variable Y].	Verdant: this will probably be searchable by variable name, but the user will need	Partially

	<p>to do some searching. Verdant can't just answer that question in its current format</p>	
Effects of [insert names of variable X1, variable X2, and variable X3] on [insert name of variable Y]	Same as above	Partially
Effects of [insert names of variable X1, X2, and X1*X2] on [insert name of variable Y].	Same as above	Partially
List of all significant predictors of [insert name of variable Y]	Verdant: they can probably figure this out by looking through all analyses, but it will take some time. Verdant cannot answer this question directly	A little
The most significant predictor of [insert name of variable Y].	Same as above	A little
Best fitting linear regression model.	Verdant: in practice this is probably one of the later regressions they did, so it should be reasonably easy to search for. However, it may take quite a bit of search if there are a lot of different models	Partially
List of all non-significant predictors of [insert name of variable Y].	Verdant: they can probably figure this out by looking through all analyses, but it will take some time. Verdant cannot answer this question directly	A little
Find me how I cleaned the data from start to finish	Verdant: they can probably figure this out by looking through all analyses, but it will take some time. Verdant	A little

	cannot answer this question directly	
What analyses did I run using <this> variable?	Verdant: this question seems pretty searchable by looking up the variable name, Verdant should return all analyses pretty directly	Fully
What analyses did I run where the difference was significant/marginal?	Verdant: they can probably figure this out by looking through all analyses, but it will take some time. Verdant cannot answer this question directly	A little
Does the ordering of the main independent variable and the covariate matter, for this function call (used in finding main result)?	No, need to test that hypothesis? If they have already tested the hypothesis maybe?	N/A
What interaction analyses did I run?	Verdant: they can probably figure this out by looking through all analyses, but it will take some time. Verdant cannot answer this question directly	A little
what were all the versions of that notebook?	Verdant can deliver directly	Fully
what were the last 3 outputs of that cell? (output is a list, then you can recover the code from the output you select)?	Verdant can deliver directly	Fully
show me the logistic regression analysis I ran without any interaction variables	Verdant: if they have a specific logistic regression in mind, this should just be a matter of searching the history of that regression	Partially
show me a histogram of the residuals from the first linear regression analysis I ran	Verdant: since they have a specific regression in mind (the first one) it should be	Partially

	doable to search that regression and then locate a histogram from that point in time. It may take a little doing though	
what were the beta values from the regression I ran earlier today	Verdant: find the regression, check the beta values	Fully
what were the distributions of my variables when I ran the boosted decision tree algorithm	No, this requires history but then also another analysis on top of that, to show the distribution. If they had the distribution back then, it may be just a matter of retrieval, but if not, this won't be doable because we can't guarantee the data will be the same even if they rerun that past analysis.	Not at all
what was the final thing I did for this project?	Verdant: activity pane, sure	Fully
what did I try along the way?	Verdant: skim through ghost books using activity pane. Can't do summarization though	Partially
hmm ok I was trying a bunch of queries with different features and different options - what was my search path through these?	Verdant can show all those queries, but the user would need to look through each to figure out the path themselves	A little
what questions did I ask that didn't pan out?	Verdant has all this history, but this would be really hard to do, take brute force search	A little
which of these csv files, sql queries, and Rmd files are, like, on the ""right"" path, and which ones were mistakes?	No... Verdant might be able to provide some evidence, but especially because we're talking about data that might	Not at all

	be outside of the notebook and summarizing it, no	
Show me the previous test result for this particular dataset.	Yes, so long as the dataset is easily searchable by name in a notebook, not possible otherwise	A little
What was the most recent test result for this table?	Yes, if the table is something searchable, the most recent test result should be a quick retrieval. Saying partially because again... with data there's a chance history isn't there of the table at a certain point in time	Partially
Show me the tests that failed for March 2016 data.	Same as above	Partially
Here's a visualization I produced, let me right click on it to give me the script to produce it.	Verdant, inspector sure	Fully
What were the parameters I used to create the summary plots, so I can run different summaries and make sure the plots can be compared.	Verdant, inspector sure	Fully
Show me the analysis I did using Fiona, not the other one where I used GeoPandas.	Should be searchable using keywords for that library, but may take a long search to narrow down	Partially
Show me the quick analysis, not the one that runs for 6 hours.	No	Not at all
How long did it take to process country X, so I can estimate the time it will take for another one.	No	Not at all

What was the state of this when the data frame in <some part of the notebook> had more than 10,000 rows	Verdant, can look at the history of that data frame, but if you didn't record the row count, additional analysis on top of history might be needed	A little
What was the state of my notebook the last time that my plot had a gaussian-ish peak?	May be doable if they can browse the history of the plot and decide for themselves what looks "gaussian-ish"	Partially
Show me, visually, a history of the plots in this notebook over the last few weeks	Sure, just search plot to retrieve them all	Fully
What data sources have I been using over the last month?	Again, since it involves data, this is dicey	A little
What are the model parameters I used for the random forest I ran on July 5, 2017?	Look at the model history, should be searchable from date and model	Fully
Show me the histogram I ran for the elastic net I ran on July 5 2017.	Should be searchable by model and date	Fully
Show me what my model accuracy looked like before I downsampled to 60Hz.	Should be searchable by the specific 60Hz	Fully
Show me the highest accuracy model I ran on July 5, 2017.	Can show all the models from that date, but they will have to look through them all	Partially
What was the AUC of my model named 'model65'?	Should be directly searchable	Fully
Show me the data frame that had the following pieces of information?	May be searchable by keywords to find that dataframe	Partially

which of the analyses had the best ROC?	Will need to look through all the models manually to decide this	Partially
Show me all the different ways I oversampled the minority class	This entirely depends on how searchable the oversampling is	A little
WHich train/test split resulted in the best P-R?	May need a pretty exhaustive search through P/R and split	Partially
restore the version of data from August 5th	No, it's data	Not at all
redo the code from August 4th, but using new data	This should be just a matter of retrieving a specific version to rerun	Fully
How many notebooks did I collect from GitHub?	It depends on if they have this number readily available in history, which they probably would?	Partially
What is the average size of code cells in Python notebooks? What about Julia notebooks?	If this is an analysis they did, sure... if not no	A little
What is the repository of the notebook that has the biggest amount of cells?	If this is an analysis they did, sure... if not no	A little
What date did miscorrelations happen?	No, would take exhaustive search through way too many things, and even then it depends on if miscorrelation was documented back in time	A little
How much miscorrelation was there?	No	Not at all

For this interesting correlation result, what were the input parameters.	Since they have a specific result in mind, this should be a simple history retrieval	Fully
patient treating system	I don't know what this	N/A
what was the series of functions I called (after extensive debugging) that I used during the first round of data analysis?	No, since we don't collect execution order, if no output was recorded this may be impossible to find	Not at all
why didn't I save this in script so I could reuse later?	No, this is a personal question	N/A
give me all lines where I used library x with parameter y;	Seems searchable, but it depends on how many keywords it would take to identify library x	Partially
have any of the libraries/ data sources I used in this project been updated since I last used them?	No	Not at all
show me similar notebooks based on either on code snippets or a sample notebook I provide (locally or via URL);	Not a history Question	N/A
who has used the same API or dataset in Jupyter notebooks (either in public ones or those that are part of the same environment, e.g. within a JupyterHub instance);	Not a history Question	N/A
show me the revisions that changed the most/ least number of lines/ characters (perhaps distinguishing between code and comments);	No	Not at all
give me a list of the most popular parameters for a given library, along with usage examples;	Not a history Question	N/A

give me punchcard stats as on GitHub	Yes, you can pretty easily see this in the activity visualization	Partially
Replicate the analysis from last Wednesday	Partially, only because of the data issue	Partially
Which HTTP queries failed?	You could answer this by looking at the history of output from the HTTP queries	Partially
Which JSON objects were not directly readable?	same as above	Partially
Which records were paginated?	This entirely depends on if pagination was recorded in history	A little
Which records returned errors?	same as above	A little
Which records had missing values	same as above	A little
latitude longitude names previous groupings	These seem like direct keywords to search, so sure	Fully
Show me all the (completed) previous analysis results for this dataset	Partially, only because of the data issue	Partially
show me how cost/profit changed during the years of a certain product	As long as this was an analysis they did at some point	A little
how did I generate plot 5	Sure, look at plot history	Fully
how did I get student names	Sure, look at history of student name list	Fully
why did this produce figure 5 and not 6	Sure, look at plot history	Partially

sniff the data	Not sure what this is	N/A
hdf5	Not sure what this is	N/A
orders cancels	Not sure what this is	N/A
read_excel	Not sure what this is	N/A
glob	Not sure what this is	N/A
Pull up script, input data, output visualizations/tables for a particular date or version of a notebook	Yes this is what Verdant does	Fully
Please go back 5 hours	Activity pane	Fully
Please revert to the state prior to the crash	Activity pane	Fully
Please give me the last packages that worked in this cell	Would need to look through cell history to figure out	Partially
compare teacher article views this year and last year in the same time frame	As long as this was an analysis they did at some point	A little
do teachers use our app during school hours or after school hours	As long as this was an analysis they did at some point	A little
How much student activity is independent, versus assigned	As long as this was an analysis they did at some point	A little
how do articles assigned by teachers differ from those that students choose for themselves?	As long as this was an analysis they did at some point	A little

Are there any specific teacher activities that tend to predict license renewals	As long as this was an analysis they did at some point	A little
How can we tell when a teacher has ""churned"" that is disengaged	As long as this was an analysis they did at some point	A little
and are there any activities we can do to bring them back?	As long as this was an analysis they did at some point	A little
Show me my experiment with the best results	No, would need summarization	A little
show me my experiment from July 5th 2017 at 1:37pm	Yes, can retrieve that exact result	Fully
What did I change in the code before the visualization broke?	Yes, backtrack using ghost book with diff highlighting	Fully
Are there any fields in the table that aren't being used in the visualization?	Not a history Question	N/A
Which part of the analysis runs the longest?	No	Not at all
project bank credit model	Yes, can retrieve history of that model	Fully
Show me the first version of the notebook	Sure, Activity pane	Fully
Training accuracy with one layer	Sure, get model history to get back to that state and look at the result	Fully
Testing accuracy with one layer	same as above	Fully
Accuracy of convolutional model	same as above	Fully

Accuracy of recurrent model	same as above	Fully
When did I generate this plot?	sure, get history of plot in artifact detail pane or search	Fully
show me the most recent analysis I published on this topic.	No	Not at all
Show me the analysis that I published that got the most engagements/comments from colleagues.	No	Not at all
Show me analyses I've done using X framework (where X is e.g. folium, shapely, etc).	May be hard to do, depends on how many keywords it takes to figure out which framework was used	Partially
show the analysis with the bar chart	Will need to search bar chart	Fully
show me the analysis that used imshow	Search for imshow	Fully
show me the analyses that used data from source X (which could be a table name, a csv, etc).	It entirely depends on the data, but here they are referencing a name, so it may be searchable	Partially
Show me the analyses of the different subsets	It depends on how searchable this analysis is	Partially