

Variolite: Supporting Exploratory Programming by Data Scientists

Mary Beth Kery
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
mkery@cs.cmu.edu

Amber Horvath
Oregon State University
Corvallis, Oregon, USA
horvatha@oregonstate.edu

Brad Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA
bam@cs.cmu.edu

ABSTRACT

How do people ideate through code? Using semi-structured interviews and a survey, we studied data scientists who program, often with small scripts, to experiment with data. These studies show that data scientists frequently code new analysis ideas by building off of their code from a previous idea. They often rely on informal versioning interactions like 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. However, data scientists must maintain a strong mental map of their code in order to distinguish versions, leading to errors and confusion. We explore the needs for improving version control tools for exploratory tasks, and demonstrate a tool for lightweight local versioning, called Variolite, which programmers found usable and desirable in a preliminary usability study.

Author Keywords

End-User Programming; Version Control Systems (VCS); Exploratory Data Analysis; Variants; Variations

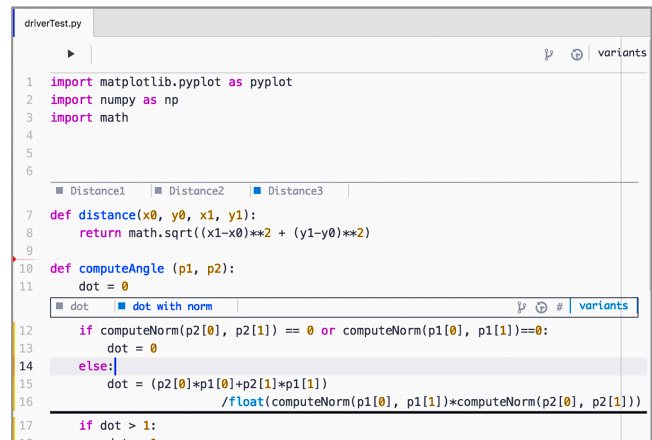
ACM Classification Keywords

D.2.3 Coding Tools and Techniques: *Program editors*;
D.2.7 Distribution, Maintenance, and Enhancement: *Version control*.

INTRODUCTION

When programmers write code to design, discover, or explore ideas, there may be no clear requirements for that code at the onset, and there may be a broad space of possible solutions [6][16]. For example, developing a new algorithm for computational biology may take considerable

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org. CHI 2017, May 06-11, 2017, Denver, CO, USA © 2017 ACM. ISBN 978-1-4503-4655-9/17/05...\$15.00 DOI: <http://dx.doi.org/10.1145/3025453.3025626>



```
driverTest.py
1 import matplotlib.pyplot as pyplot
2 import numpy as np
3 import math
4
5
6
7 def distance(x0, y0, x1, y1):
8     return math.sqrt((x1-x0)**2 + (y1-y0)**2)
9
10 def computeAngle (p1, p2):
11     dot = 0
12     if computeNorm(p2[0], p2[1]) == 0 or computeNorm(p1[0], p1[1])==0:
13         dot = 0
14     else:
15         dot = (p2[0]*p1[0]+p2[1]*p1[1])
16             /float(computeNorm(p1[0], p1[1])*computeNorm(p2[0], p2[1]))
17     if dot > 1:
18         dot = 1
```

Figure 1. Variolite is a code editing tool that includes local versioning of chunks of code. Here, there are two version boxes. The outer one has three “Distance” versions, and the inner one has two “dot” versions with “dot with norm” currently being used.

trial-and-error on both the code and the concepts behind it [35]. We call this process *exploratory programming* [32][5], which we define as a programming task in which a specific goal or a means to that goal must be discovered by iteratively writing code for multiple ideas.

Data scientists are a group that does a lot of exploratory programming. The term “data scientist” has a broad [29] and sometimes contested definition [21] but here we use “data scientist” in the simple sense of people who manipulate data to gain insights from it. We specifically target data scientists who write code, which is a large group [29] encompassing people who work with data in domains such as engineering, design, business, and research. One example of when data scientists engage in exploratory programming is “exploratory data analysis,” which is a common approach to visualizing and asking questions of data, rather than more straightforward hypothesis testing [12]. Another example is working with data to develop machine learning models or other intelligent systems, which is often a painstaking process of experimenting with different manipulations, parameters, and algorithms [7][14].

A great deal of prior work on programming tools, such as live programming tools and development environments like

R Studio, has focused on making it easier to quickly prototype ideas [25][24]. Other literate programming tools, such as Knitr [30] or Jupyter notebooks [8] have become popular in recent years for clearly communicating data analysis code, rationale, and output. Yet there is little work on how data scientists manage an abundance of exploratory code over a longer-term project. How do data scientists transition from one data manipulation idea to the next? To what degree do they compare alternative ideas, or revisit older ideas?

Current tools have limited support for variations and versions of ideas. Live programming environments, whose strength is instant feedback, tend to de-emphasize history, which DeLine et al. remarked was an issue their users cared about [25]. Tools like Jupyter notebooks have become immensely popular for data science work [9], however the focus of literate programming tools is to clearly communicate analysis work, so that it can be shared with others and replicated. We found that data scientists frequently try out many less successful paths before cleaning up and condensing their final result into code to publish or share. The linear, narrative style of notebooks gives limited help with the messy and nonlinear path to the final result.

Understanding how humans approach the exploratory process of data science is crucial for improving the usability and accuracy of this kind of work. Patel et al. [14] 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’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*” [14]. Others who have studied data science and machine learning developers, such as Hill [7] and Guo [22] 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 [13][16][34]. Further, when even experts struggle with an exploratory process, this lowers the accessibility to novices programming in these domains.

To better understand the barriers and opportunities in this area, we conducted semi-structured interviews with 10 data scientists, and then a survey of an additional 60 data scientists, specifically focused on how ideation is carried out at the concrete level of artifacts like source code and files. In summary, we found that data scientists frequently write code for new analysis ideas by repurposing or building off of their code from a previous idea. Due to the sometimes non-linear progression of their exploratory work, partici-

pants emphasized reusing code, copying code and keeping code that was sometimes long dead, just in case it became useful later. To facilitate reusing code that overlapped with older analyses while attempting to keep those older analyses intact, many relied on ad-hoc means to version their code. These informal versioning practices took the form of copying files, copying snippets, and comments (sometimes with intricate combinations of code commented on and off).

While these code versioning needs seem like they could use standard software version control systems (VCSs) like Git or SVN, we note: 1) a low usage of VCSs even by participants who actively used one for other kinds of work, and 2) benefits in informal versioning interactions that a conventional VCS does not provide. For example, copying and commenting out code allows for rapid branching and comparison between multiple versions of a single line of code, or any size code of interest, without leaving the code editor. Informal versioning is more transparent than versions of code hidden behind a conventional command line tool, since in that case programmers cannot quickly view and swap among multiple versions of code. However, participants also reported consistent difficulties that arose from informal versioning practices.

We found these ad-hoc interactions are practices individuals had developed for themselves over time and experience. Yet there was considerable overlap in the practices participants reported. Commenting, copying, and keeping code are natural affordances of text programs, and require no new tools. Our studies showed that many people leverage these mechanisms for a variety of purposes, like versioning. By developing a stronger understanding of the workarounds programmers use, we aim to explore what versioning capabilities data scientists need for managing their ideas in a non-linear exploratory process. Through design, we aim to leverage the local interactions that programmers naturally use during exploratory programming, such as versioning arbitrarily sized sections of code.

Our new tool, called Variolite ¹(see Figure 1) is an initial probe to test some of these interaction ideas. Variolite, which is a kind of rock structure, here stands for Variations Augment Real Iterative Outcomes Letting Information Trascend Exploration. Variolite is built on top of the Atom editor [1], and allows users to directly version code in their editor simply by drawing a box around a section of code and invoking a command. It is a general purpose, programming-language agnostic, versioning tool. We performed an initial usability study that showed that it is usable.

Our work makes contributions in two areas:

- Our qualitative study showing barriers and requirements of data scientists managing exploratory code.

¹ Variolite source code <https://mkery.github.io/Variolite/>

- A working prototype of a tool to address some of these versioning interaction issues, and a usability study showing its usability.

RELATED WORK

Programming for data analysis or machine learning

Guo previously studied people who program to gain insights from data, whom he called “research programmers” [22]. He noted problems managing a large amount of files produced during exploratory work with data. He also noted a “lack of transparency” in traditional versioning tools, in contrast to the “low-tech” interaction of copying a file and giving it a derivative name (like “file-2”). We expand upon this work by studying a wider range of informal versioning interactions through interviews and through looking at interviewees’ project artifacts.

Guo et al. [23] also produced a research system called Burrito, which displays a GUI activity feed of things like outputs, save events, and notes relevant to a given project. While both Burrito and Variolite record history and output provenance information, the Burrito tool collects much more detailed provenance information by working at the operating system level in Linux. Variolite is much more lightweight and is situated in the code editor, but as a trade-off, collects a much simpler link between code and output. Burrito is also intended as a lab notebook approach, like literate programming tools such as Jupyter notebooks [8]. While Variolite certainly could work with added Markdown and other literate programming techniques, our focus in this research is managing ideas during non-linear exploration.

Both Hill et al. [7] and Patel [15] reported that machine learning programmers struggled to reproduce or understand earlier experiments. Patel found that logging experiments was beneficial to return to earlier result, but difficult and time consuming. Patel created a research tool 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. Hindsight is a GUI based tool specific to classification. Variolite aims to generalize interactions for dealing with alternatives and history to work in a wider range of exploratory tasks.

Dealing with alternatives

Several other research tools have explored interactions for alternatives of code, and also versions of code over time.

Juxtapose [6] is a research tool that provided interaction designers with different alternatives of their code, in order to compare between 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. [18]. Juxtapose also built off of prior work such as Set Based Interactions [17] and Subjunctive Interfaces [4], which explored general techniques for exploring multiple

alternatives in parallel. These were not specific to writing programs.

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 [20]. Software product line research aims to handle much more complex versions and interdependencies than Variolite, with commensurate complexities in the developer’s interface.

Interacting with history

Azurite [33] developed an interaction 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 [27]. None of these focused on helping users edit fine grain versions.

METHODOLOGY

Before we approached the design of a tool, we wanted to understand the real needs and barriers faced by data scientists, so we performed two formative studies.

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, and 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 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 [16].

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 [10]. 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 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 2. 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 encounter, 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.

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?” - P6

This sentiment is common to how end-user developers prioritize goals [3]. While “end-user developer” often refers to a programmer without formal training in computer science, many of our participants did have formal training (Figure 2). 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 [3]. 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 “Of-

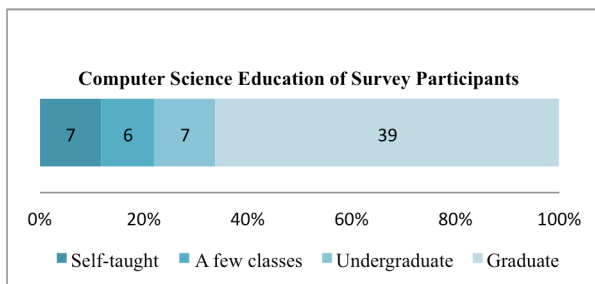


Figure 2. Background statistics about 59 of the survey respondents. Most had graduate degrees in computer science and work in research (one respondent declined to answer).

ten”. 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 non-linear 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.” - P9

Code that is commented out does not run. Using comments to store code has been observed in prior studies [6][31]. Ko et al. [2], 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.

P3’s code, shown in Figure 3, 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 a second graphing section of code lower down that is fully commented out.

Duplicating snippets, function and files

Copying code was another popular way of versioning. Shown in Figure 4 is P1’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 “Occasional-

ly” 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.

```
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. Comments used to keep alternatives

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 4). 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? 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! P1 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 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, although 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 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 [16]. Nguyen-Hoan et al., in a survey of scientists, found that 30% of their sample used VCS [16]. Despite our over sampling of people with CS backgrounds in our survey (see Figure 2), we found that 48% of participants, just under a half, did *not* use version control for their exploratory data analysis work.

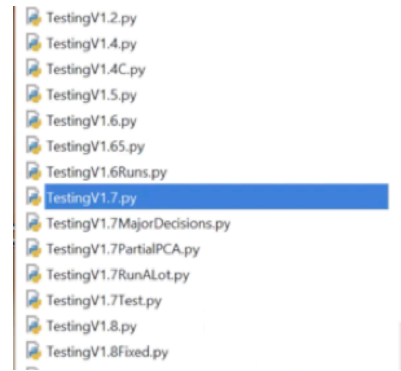


Figure 4. Folder from a data analysis project of P1

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” - P5

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.



Figure 5. Creating a variant box in Variolite. In (a) the user selects a section of code and selects the command “wrap in variant”. In (b) this places a box around the code, which can be used to keep multiple versions of that chunk of code (c, d).

DESIGNING TO SUPPORT LOCAL VERSIONING

To explore this design space, we first sketched a number of possibilities for interacting with versioning 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.

Based on this feedback, we iteratively designed and implemented a prototype tool called Variolite. Variolite is implemented in CoffeeScript and CSS, using the Atom editor’s package framework [1]. 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. Although Variolite 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.

Users of Variolite draw “variant boxes” around regions of code, where the code within the box can then be locally versioned or branched (Figure 5). 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 [6]. 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 b,c). Variant boxes can be created or dissolved back into flat code as needed.

Variolite Use Case

Interviewee P1 was working on developing a machine learning model. To do this, she created many different copies of her Python scripts, in order to keep a record of all the variations of the code while trying to achieve the highest predictive accuracy for her model. She also developed an elaborate menu system in her main script, to experiment with different combinations of aspects like algorithms, features, and which features were used in the model.

To illustrate the range of functionality of Variolite, we will describe a fictional use case, based on how P1 might use it. The name “Ellen” and domain in the scenario are fictional:

Drawing a box, version it

Ellen is calculating a new feature for her model. She has a function `matchString()` and is trying to figure out how to best measure similarity between two pieces of text. She wants to try a new method for doing this. Using Variolite, she selects the code of `matchString()` with the cursor and selects the command “Wrap in Variant” from the right click menu (Figure 5a). This draws a *variant box* around `matchString()` (Figure 5b). Now Ellen uses the “new branch” menu item to create a new version of `matchString()` (Figure 5c), which she titles “fuzzy match”. She edits this new version to implement the new algorithm. She can now switch between the two algorithms of `matchString()` by changing the active tab in the header of the variant box (Figure 5d). Whichever version is showing is the one that is run.

Supporting nonlinear exploration

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. As Ellen works, she can vary different aspects of her code. She puts a variant around a single line that changes which machine learning algorithm she is using. She puts variants around different functions that generate features and that control which features are included in the model. Now, by simply interchanging which versions of each of those alternatives are run, she can explore different *combinations* of the different things she has tried. Instead of a purely linear iteration, she can re-try features she used in the past with new versions of the algorithms’ parameters. This allows Ellen to more easily explore a problem that can be improved on multiple interrelated dimensions.

Running code

Switching version tabs does introduce a danger of runtime errors if surrounding code that a version was written in has since changed. In future work, this limitation may be addressed using lightweight program analysis, e.g. by at least alerting users to parts of the code that are likely to produce errors.

As Ellen works, she can run her program using the run button at the top (Figure 7a), or a terminal area on the right pane of Variolite (Figure 7e). This will run the entire file at its current state. As with a typical terminal, Variolite displays the command used and output from each run in its terminal pane. By simulating ways to run code within Variolite’s interface, we are able to add layers of interaction for dealing with runs. Exploration with data involves not just the code artifact, but also the data and parameters used to produce a certain result. Therefore, each time the code is run, a wrapper script in Variolite records the parameters used, and all inputs/outputs from the run. This provenance data is saved in JSON format separately from the code. However, keeping all outputs can be impractical for space when the output is very large. To mitigate this, users have the option of saving whole data files or simply pointers to those files. Saving only pointers is a trade-off in that Variolite cannot provide accurate provenance information if those data files have changed.

To allow users to revisit past experiments, a commit is also automatically recorded of the entire source code file each time the file is run, as well as a commit for each variant box used in the run. Suppose Ellen would like to review an earlier experiment. She can find the experiment’s result in the output pane, or use the search pane to find it if it is out of view (Figure 7d). To help users find interesting results among many runs, a user can provide annotations and tags on versions that are promising, either in the variant box, or in the output panel at the right (Figure 7f).

Double-clicking a given output 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 the past. While viewing an earlier commit, Ellen cannot edit the code, but she 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 Ellen creates a branch from a past commit, however, she can continue editing from that point.

If Ellen wishes to backtrack a single variant box to an earlier state, she can also navigate its commit data by clicking the clock icon (Figure 6). 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.

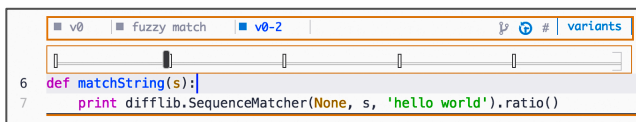


Figure 6. Navigating to a past commit. The orange color of the variant box indicates that the code is showing a past state.

Automatic commits also serve as a safeguard for backtracking. In the earlier example, suppose Ellen decides to keep

only the one string-matching algorithm as the best solution. She presses the button “Dissolve Variant” from the box header, which dissolves the variant box around `matchString()` and returns her code to flat text. However, the variant and the other versions of `matchString()` are not really gone, since they can be retrieved using undo in the short term. Ellen can also use the commit history of the code to retrieve old versions of `matchString()` even months later, since that variant box will still exist in earlier commits of the file. This reflects the finding that people do not always know in advance what they will find useful [32].

Modeling alternatives and history

In the same sense as with a normal VCS, creating a new tab in a variant box creates a new *branch* of the code in that box. In a VCS, a *commit* is a copy of source code taken at a certain point in time, and a series of commits are a way to track the history of the software’s development. A *branch* allows for multiple concurrent copies of the software’s code. 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. Shown as a white header bar at the top of the file (Figure 7a), this top-level variant box wraps the entire file and also acts as a general menu for Variolite, where the user can run the code.

We provide this approach to branches and commits in order to minimize the upfront investment users must take to think about their code’s history. In a simple use case, similar to Juxtapose [6], the user does not have to think about versioning more than switching between tabs. However in Variolite we consider that data scientists may work on code over the span of weeks or months. Thus to support long term versioning needs, the full capabilities of a detailed revision tree are available to users if they need it. This is in line with the “low-floor, high-ceiling” principle proposed for creativity support tools [19]. Currently in the prototype of Variolite, all revision data is kept in JSON files separately from the code. This is brittle to external edits, a common round-trip engineering problem. In future work, Variolite can compare between the last recorded version of the file and the one opened by the user, and interact with the user to resolve ambiguities over where to place variant boxes if the annotations are broken. In future work, we will investigate storing the revisions in a way that is consistent with Git.

MANAGING MANY VERSIONS

A chief design concern in creating an automated alternative for informal versioning practices is simply “moving the problem” instead of addressing the difficulties data scientists have with informal versioning. However, unlike informal practices, as programmers scale up the number and

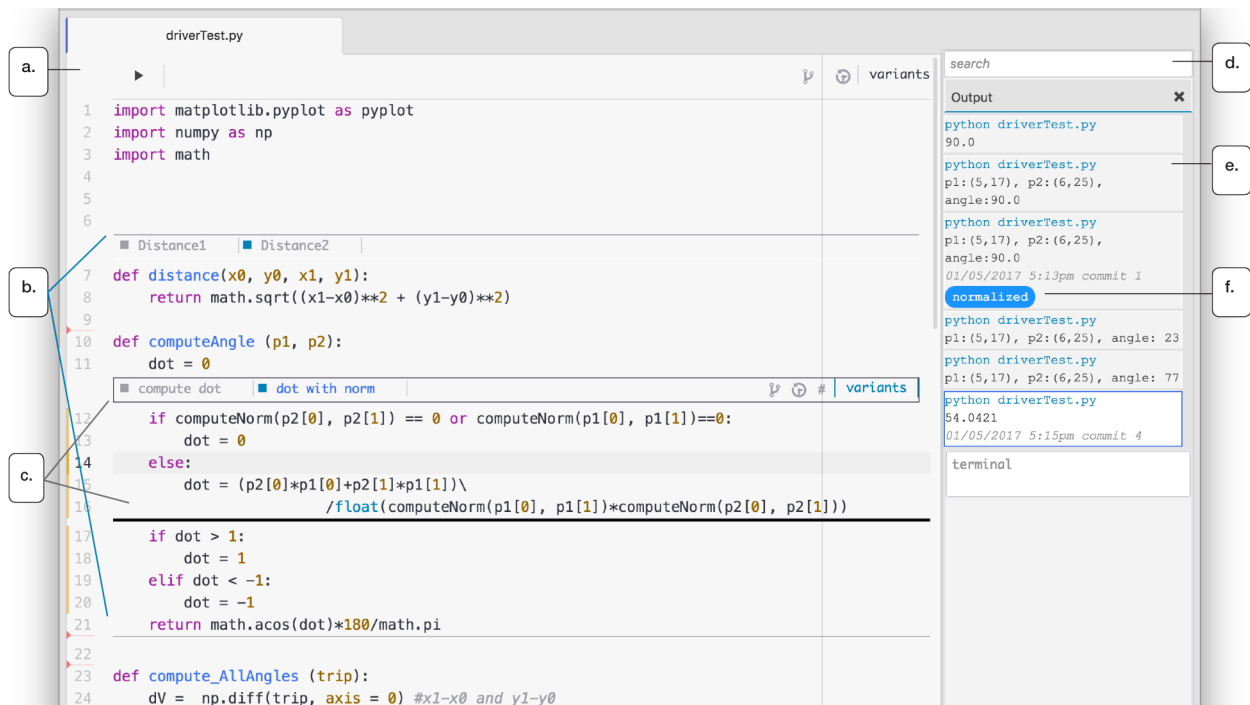


Figure 7. Variolite with labels for different features. (a) the top level variant box that wraps the entire file and also acts as the tool menu. (b and c) two different variant boxes, one nested within the other. (d) a search bar for finding outputs and versions. (e) the output pane and (f) where the user has given an output and its commit a custom tag.

complexity of versions they choose to keep, we can use design interventions in a tool to help with these issues.

Code readability

Variolite can improve code readability by reducing the number of loose copies and commented-out sections of code. Yet a design trade-off of situating versioning in the editor is that the GUI components of Variolite must not obstruct code readability. This problem may be amplified when the user has multiple variants throughout their file. We address this problem by styling variants as much as possible like in-line comments. As shown in Figure 7 b and c, variant boxes display a header bar with different controls when the user’s cursor is placed within the box. When the cursor is outside, the box is displayed as a line in the same style as a code comment.

Second, 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. Still, 3 or 4 branches may be reasonable much of the time, as three is the average number of *file*-copied versions that survey participants reported making. Limiting the space is another design tradeoff for readability. A list of 10 to 20 branches in a variant box may look overwhelming. As a compromise, in Variolite the user can access a larger revision tree showing all the branches of that box (Figure 8), such that they can control which 3 or 4 branches are actively showing.

Distinguishing versions

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 Srinivasa et al. [28] 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.

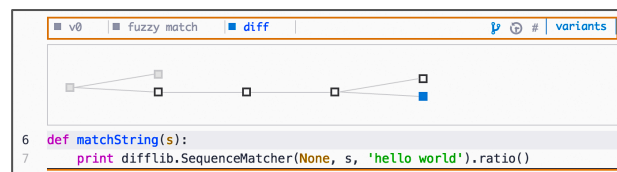


Figure 8. Navigating branches

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 7f).

- A snapshot of the latest output for each branch and each commit. Currently, the captured input/output is only from the console, but capturing pictures, like displayed charts and graphs, are planned as future work.
- The ability to search in past outputs, branches, and commits and not just in the current file (Figure 7d).

Although cues may help, the complexity of interacting with local versions may be a general limitation of a local versioning tool. Variolite may be convenient with smaller scripts that data scientists often use. However we cannot yet claim that it scales to large projects. However, we argue that informal versioning techniques like copying/commenting also do not scale well to large projects. Local versioning 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 [20]. The focus of Variolite is using interaction techniques and exploring designs to mitigate this complexity, but having files with variability is a known usability challenge, which we will continue to research.

PRELIMINARY USABILITY STUDY

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 pilot usability study. To test the underlying variant box interaction, we limited the Variolite prototype to only variant boxes with tabs (Figure 7b). The additional features reported above were added afterwards using feedback from the usability pilot.

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 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 may 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 described above, 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 8).

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.

FUTURE WORK

While we have determined the basic usability of key features of Variolite, a next step is to evaluate the usefulness of all of these interventions together. To do this, we are preparing the tool for longer-term usage by data scientists. A study over days or weeks will allow us to explore how data scientists use versioning, as well as how Variolite’s interactions scale and work over time.

CONCLUSIONS

Data scientists must manage complex ideas over time during exploratory programming work. Exploration is iterative and sometimes non-linear. Data scientists often resort to informal methods of versioning, such as commenting out code, in order to keep around old attempts to return to or compare. These informal versioning methods are widespread, and cause difficulties and confusion when the programmer cannot keep a strong mental map of their code. Variolite shows that new interactions for version control can begin to address these difficulties. Creative exploration is key to many kinds of work, as is managing a variety of ideas. By closely studying people’s workarounds and how people *want* to use versioning, we can design usable interfaces for software versioning tools for a wider variety of people and a wider variety of tasks. Variolite’s lightweight local versioning ideas may also be useful for other creative editing tools, such as text or image editing.

ACKNOWLEDGEMENTS

This research was funded in part by the NSF under grants IIS-1314356, IIS-1644604 and CCF-1560137. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the US Government. In addition, we would like to thank Kenneth Holstein and all our participants.

REFERENCES

1. Atom Editor, <https://atom.io/>
2. Andrew 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.
3. Andrew J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, 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.
4. 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.
5. Beau Sheil. 1983. *Environments for exploratory programming*. Datamation 29, 7 (1983), 131–144.
6. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. 2008. *Design as exploration: creating interface alternatives through parallel authoring and runtime tuning*. In Proceedings of the 21st annual ACM symposium on User interface software and technology. ACM, 91–100.
7. Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. 2016. *Trials and Tribulations of Developers of Intelligent Systems: A Field Study*. In 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, 162–170.
8. Fernando Pérez and Brian E Granger. 2007. *IPython: a system for interactive scientific computing*. Computing in Science & Engineering 9, 3 (2007), 21–29.
9. Helen Shen and others. 2014. *Interactive notebooks: Sharing the code*. Nature 515, 7525 (2014), 151–152.
10. Hugh Beyer and Karen Holtzblatt. 1997. *Contextual design: defining customer-centered systems*. Elsevier.
11. John M Carroll and Mary Beth Rosson. 1987. *Paradox of the active user*. The MIT Press.
12. John W Tukey. 1977. *Exploratory data analysis* (1st, ed.). Pearson, New York, NY.
13. 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.
14. 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.
15. Kayur Dushyant Patel. 2013. *Lowering the Barrier to Applying Machine Learning*. Ph.D. Dissertation.
16. 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.
17. Michael Terry, Elizabeth D Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2004. *Variation in element and action: supporting simultaneous development of alternative solutions*. In Proceedings of the SIGCHI conference on Human factors in computing systems. ACM, 711–718.
18. 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.
19. Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, Ted Selker, and Mike Eisenberg. 2005. *Design principles for tools to support creative thinking*. (2005).
20. Paul Clements and Linda Northrop. 2001. *Software product lines: Patterns and practice*. Boston, MA, EUA: Addison Wesley Longman Publishing Co (2001).
21. 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>
22. Philip Jia Guo. 2012. *Software tools to facilitate research programming*. Ph.D. Dissertation. Stanford University.
23. 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.
24. R Studio <https://www.rstudio.com/>
25. 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.
26. Robert Hawley. 1987. *Artificial intelligence programming environments*. Intellect Books.
27. Sandeep Kaur Kuttal, Anita Sarma, Amanda Swearngin, and Gregg Rothermel. 2011. *Versioning for Mashups—An Exploratory Study*. In International Symposium on End User Development. Springer, 25–41.

28. 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.
29. Thomas H Davenport and DJ Patil. 2012. *Data scientist*. Harvard business review 90 (2012), 70–76.
30. Knitr tool, <http://yihui.name/knitr/>
31. 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.
32. YoungSeok Yoon and Brad A Myers. 2014. *A longitudinal study of programmers' backtracking*. In 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, 101–108.
33. 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.
34. Zeeya Merali. 2010. *Computational science: Error, why scientific programming does not compute*. Nature 467, 7317 (2010), 775–777.
35. 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.