ABSTRACT

SHRESTHA, NISCHAL. Supporting Just-In-Time Learning for Data Science Programming. (Under the direction of Christopher Parnin.)

Data science programming presents many challenges for programmers entering the field. Roughly, data science programming can be broken up into several activities: data wrangling, analysis, modeling, or visualization. Data wrangling is an important first step that involves cleaning and shaping tabular data—or dataframes—into a form amenable for conducting analysis. However, data wrangling code is challenging because it involves learning a plethora of data transformation operations and how they can be composed together to shape the data. Data wrangling code requires tracking and understanding numerous data transformation techniques, and it is a tedious and error-prone process. Prior work has mainly focused on tools that help end users and programmers wrangle data by providing better management of code in computational notebooks or through GUI tools that attempt to remove the need to program. However, there is a gap in the literature and existing tools to support programmers in understanding, exploring, and debugging data wrangling code interactively and flexibly. The thesis of this dissertation is: Programmers can understand, explore, and debug data wrangling code flexibly when aided by just-in-time learning tools that accommodate multiple learning objectives.

The goal of this research is to help programmers understand, explore, and debug data wrangling code by exploring two just-in-time learning tools. The first study provided evidence that programmers heavily rely on opportunistic learning strategies, which involves using quicker resources and learning topics as needed. We also found that learning a language involves adapting to an entire ecosystem which includes libraries, tools, and the community. In the second study, we investigated how an online community of practice can help data scientists in the R community through a social coding project called #TidyTuesday on Twitter. We found that an online community of practice provides motivation, dissemination of knowledge, and adoption of best practices. A community of practice is a just-in-time learning tool that provides programmers flexibility on what they want to learn by browsing, adapting, and extending others' code. To help programmers understand and explore data wrangling code, we built Unralve, another just-in-time learning tool for the RStudio IDE (Interactive Development Environment) that presents visual cues and summaries of data transformations, and enables exploration via simple structured editing of the code. In a formative study, we found that Unravel provides diverse learning activities such as discovering code behavior, relationships between functions, and exploring code alternatives. To help programmers learn about and debug problems in data wrangling code effectively, we extended Unravel to highlight problems about the code and data through always-on visualizations and and automate data quality checks.

Supporting Just-In-Time Learning for Data Science Programming

by
Nischal Shrestha

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2022

APPROVED BY:

Kathryn Stolee	Tiffany Barnes
Shiyan Jiang	Rob DeLine

Christopher Parnin
Chair of Advisory Committee

DEDICATION

I dedicate this dissertation to my mother, my father, and my brother. None of this would have been possible without their love and support in all of my endeavors.

BIOGRAPHY

Nischal Shrestha was born in Kathmandu, Nepal in 1992. He immigrated with his family to Chapel Hill, North Carolina, USA in 2002. Nischal went to elementary through high school in Chapel Hill. Before college, he knew he wanted to do something in engineering so he applied to North Carolina State University (NCSU).

He studied his undergraduate at NCSU from 2011-2015 earning his Bachelor of Science in computer science. Nischal then interned at WillowTree Apps before he started graduate school at the same University. This experienced piqued his interest in software engineering research.

In graduate school, Nischal was interested in issues related to both software engineering (SE) and human-computer interaction (HCI). As he waded through the first and second year, his interest grew towards issues related to learning programming languages. He joined the Alt-Code lab to work on this topic under the guidance of Dr. Chris Parnin. There, Nischal learned how to conduct SE research with a flavor of HCI and published a few papers around data science programming and discover ways to help programmers wrangle data more effectively. Nischal eventually interned at RStudio during his 5th year, which shaped his thoughts and the tools he used for the last two projects for the dissertation.

Beyond work, Nischal has always maintained an interest in music and powerlifting. Although he has not yet competed professionally, he can be seen hoisting some barbells and weights at the gym. Nischal also enjoys both listening to and making music through guitar, bass and through Sonic Pi on the computer.

ACKNOWLEDGEMENTS

In the last several years as I worked on my PhD, I have been blessed with a lot of peers, colleagues, mentors, and friends who have helped me throughout my journey.

First, I would like to thank Chris Parnin for his support regarding my research and career development. Chris has always helped me push past barriers, move fast, and continually develop both academic and industry-relevant skills. This work would not have been possible without Chris.

I want to thank some great mentors and colleagues I have had the privilege of working with. I want to thank Titus for guiding me on research writing, collaborating with me on most of my research projects, and just being a great colleague to work with throughout the years. I would also like to thank Denae who I always looked up to as inspiration of conducting human-computer interaction research, and who always provided guidance on all things career related. Additional thanks to the rest of the Alt-Code lab. Thank you Eric, Mahnaz, and Samim who have always provided me with great feedback on my work, encouraged me when I felt overwhelmed, and being there for me throughout the pandemic. The Software Engineering lab was in many ways the perfect space for me to grow as a PhD student so I want to thank the following peers who have helped me along the way: Justin Smith and Justin Middleton, Chris Brown, Gina, Kai, and George. Finally, I want to also thank Greg Wilson and Alison Hill who both provided helpful advice and ideas on my research, and who also shaped my career along the way as I waded into the data science world.

Lastly, I have made some amazing friends along the way that have helped me in many different ways. First, thank you Huy Tu for being a great friend, keeping me in check with work-life "balance", introducing me to Vietnamese food, and making grad school more fun by inviting me to all the festivities. Thank you to my "beer hang" troupe, who I could commiserate with while enjoying some beers: Andrew, Kenneth, Adam, Justin, Nick, Shrikanth and Fogo. I want to also thank some friends I met from other universities like Katie, Rebecca, and Mariam who have been supportive in all of my research endeavors. Finally, I want to thank some of my fellow RStudio interns: Thank you Daniel, Maya, and Simon for nerding out about R with me and helping me with my research.

I want to also thank some of my non-PhD friends who have supported me over the years. Thank you Mylo, I probably couldn't have endured all of the harder points of my journey without your never-ending support and friendship, as well as the "wild boyzzz", Opti, Bravo, and Sarge. I also want to thank Henry, Noel, and Carlos for encouraging me to keep going when things felt slow.

Funding

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 1559593, 1755762, 1814798, and 2006947.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

THESIS

Data wrangling is an important step in data science programming that requires data to be transformed into a form amenable for analysis. However, data wrangling is a time-consuming and error-prone process that requires a programmer to learn and correctly apply numerous data transformation techniques. Programmers can understand, explore, and debug data wrangling code flexibly when aided by just-in-time learning tools that accommodate multiple learning objectives.

CHAPTER

2

INTRODUCTION

People across a wide range of professions now write code as part of their jobs with the purpose of obtaining insights from data rather than building software. The popular term for this type of work is "data science" and the group of people are often called "data scientists". Data scientists come from various backgrounds like engineering, business, design, and research [Dav12]. They are increasingly prevalent in both industry and academic settings. In industry, data scientists work in numerous sectors like public policy, technology, and healthcare [Loh17]. In academia, data scientists are graduate students, professors and technical staff writing code to make research discoveries [Guo12]. Data scientists are known to be like end-user programmers, writing code as a means to an end—to gain insight into data. Unlike traditional programmers, they are also a group that heavily engages in exploratory programming [Ker17c]. In particular, data scientists heavily engage in exploratory data analysis where they continually explore questions about the data and iteratively refine statistical models and visualizations to paint a story. However, data scientists also share similarities with software engineers, writing reusable analysis code to share with others. Ko et al. [Ko11] calls this *end-user software engineering*.

Despite the growth of data science, there is little understanding of how data scientists are learning and practicing their skills. Prior work has given insight on the activities data scientists engage in at work [Guo12; Kan12; Ker17c; Kim16; Rul18; Seg07], and how practitioners teach beginners in both industry and academia [Kro19]. There are also a few studies examining how data scientists hone several skills such as acquiring, cleaning, wrangling, visualizing, and presenting data [Har15; Kan12]. To gain expertise in these skills, data scientists must decide between formal and informal learning paths: data scientists could acquire a data science major [Tre17; Van18] versus taking MOOCs (massive open online courses), or participate in hands-on workshops [Wil06] and coding

bootcamps [Top; Cam20]. Although there is a trend towards lighter, and informal online learning—such as interactive tutorials [RSt20]—there is a lack of research on how to help data scientists learn programming as they work with the tool for writing code like their IDE (Integrated Development Environment).

Among the various activities in data science programming, data wrangling presents an immediate barrier for those entering the field. Data wrangling [Guo11; Kan11; Rat17; Sut18] has been described as requiring up to 80% of the time and effort in a data science project [Goe3; Kan11; Rat17]. "Dirty data" was the most commonly-reported challenge in the 2017 Kaggle survey [Kag17] and the 2020 Kaggle survey reports that "most data scientists continue to learn outside of formal education" [Kag20]. As Sutton et al. note, the problem can be exacerbated with large datasets which often present multiple problems, leading to "death by a thousand wranglings" [Sut18]. Martin reports that data science workers tend to spend less time doing analyses, and more time preparing their data: "Weeks or months is a realistic timeframe. Hours is not." [Mar18]. Engaging with data takes a lot of time and effort.

To help data scientists, this dissertation builds on existing tools and techniques to support data scientists understand, explore, and debug data wrangling code through just-in-time learning solutions. Given the exploratory nature of data science programming [Ker17b] and the trend towards informal, opportunistic learning strategies [Bra09], this work offers solutions that help data scientists as they engage with data. Prior work has largely focused on building tools that help data scientists write and manage code in computational notebooks, which is a popular coding environment that allows a mix of code, narrative, and other media like visualizations. For example, Gather [Hea19] helps analysts find, clean, recover, and compare versions of code in cluttered, inconsistent notebooks. For data wrangling, Wrangler [Kan11] is an example of an interactive tool designed to ease the process of writing data transformation scripts. However, there is a lack of tooling to help data scientists understand and explore code and its behavior. This work aims to support data scientists by helping them more easily gain insights into code behavior as they wrangle their data for analysis.

CHAPTER

3

CHALLENGES RELATED TO LEARNING PROGRAMMING LANGUAGES

Peter Norvig wrote a guide, “Python for Lisp Programmers” [Nor00], to teach Python from the perspective of Lisp. We interviewed Peter regarding this transition and he described a few challenging aspects of switching to Python such as how lists are not treated as a linked list and solutions where he previously used macros required re-thinking. When asked about the general problem of switching programming languages, he said, “Most research is on beginners learning languages. For experts, it’s quite different and we don’t know that process. We just sort of assume if you’re an expert you don’t need any help. But I think that’s not true!” Peter believes that learning new languages is difficult—even for experts—despite their previous experience working with languages. Is Peter right?

In this chapter, we investigate this question through an empirical study of Stack Overflow questions across 18 different programming languages and semi-structured interviews with professional programmers [Shr20] to further examine the diversity of challenges they face when learning new programming languages. We hypothesized that previous knowledge could potentially interfere with learning a new programming language. From our inspection of 450 Stack Overflow questions, we found 276 instances of interference that occurred due to faulty assumptions originating from knowledge about a different language. To understand why these difficulties occurred, we conducted semi-structured interviews with 16 professional programmers. The interviews revealed that programmers make failed attempts to relate a new programming language with what they already know. Our findings inform design implications for technical authors, toolsmiths, and language designers,

such as designing documentation and automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem.

3.1 Motivating Example

In psychology and neuroscience, studies have shown that confusion can occur when older information interacts with newer information [Und57; Jon98; Pos04a; Pos04b; Bad05; Jon06]. To illustrate, suppose the bread aisle of your favorite store was recently moved. You may reflexively start walking towards the old location due to *interference*—when previous knowledge disrupts recall of newly learned information. However, if you recently saw that the impossible burger was added to the frozen section (and not a separate health aisle), using knowledge that frozen food can be found in the frozen section is an example of *facilitation* [Aue17]—when previous knowledge helps retrieval of new information. In the same vein, when a Java programmer is learning Kotlin, we postulate that their prior Java knowledge either facilitates or interferes with learning. The knowledge that Java is objected-oriented and uses static typing facilitates their learning as Kotlin shares similar properties. The knowledge that Java classes are not `final` by default interferes with their learning because Kotlin classes are `final` by default.

Hypothesis: If previous programming knowledge can be framed as a source of interference with new programming language acquisition, interference theory can explain why programming language learning can be difficult for experienced programmers. And when previous programming knowledge isn't relevant, learning can also be difficult because this knowledge doesn't facilitate.

3.2 Methodology

To investigate our hypothesis, we first looked for evidence that programmers could have difficulty learning another language due to interference from their previous knowledge. To this end, we conducted an empirical study examining questions posted on a popular question-and-answer site, Stack Overflow.[1] We analyzed 450 posts for 18 different programming languages and qualitatively coded each post, characterizing posts in terms of whether or not programmers made incorrect assumptions based on their previous programming knowledge. Then, to understand what learning strategies programmers used when learning another language—and why previous knowledge could interfere with this process—we interviewed 16 professional programmers who had recently switched to a new programming language. We do so through the following research questions:

3.2.1 Research Questions

- **RQ1: Does cross-language interference occur?** We examined questions programmers had about programming languages on Stack Overflow for evidence of interference with previous

[1] `https://www.stackoverflow.com`

programming knowledge.

- **RQ2: How do experienced programmers learn new languages?** To gain a better understanding of why cross-language interference occurs, we interviewed professional programmers on how they learn new languages.
- **RQ3: What do experienced programmers find confusing in new languages?** To examine the ways in which programmers mix a new language with their previous knowledge, we asked programmers about obstacles they faced, and surprises they encountered in their new languages.

3.2.2 Phase I: Study Design for Stack Overflow

To answer RQ1, we conducted a study using Stack Overflow posts.

3.2.2.0.1 Data collection

To gather Stack Overflow questions, we used the SOTorrent [Bal18] data source from the 2019 MSR Mining Challenge. We queried 26 programming languages used previously by Erik [Ber17] and Waren [Lon17] in their investigation of popular language migrations, based on Google search keywords and Github repositories. We gathered Stack Overflow questions for each `<language A, language B>` pair. To keep the analysis tractable [Mas10], we considered only the association between the two languages, and not the direction of the possible interference. We used a stop-rule criteria to cover over 95% of total posts, which resulted in 15 out of the 26 language pairs shown in Table 3.2. The materials for the study are available online.[2]

3.2.2.0.2 Query criteria

We used BigQuery[3] to query the SOTorrent database and used the following filtering criteria to capture potential posts where the programmers are asking questions about a new language (target) coming from a previous language (source):

1. The question is tagged with both languages, or
2. The question is tagged with the source language but contains the text of the target language in the title or body, vice-versa.

3.2.2.0.3 Analysis

To understand whether or not cross-language interference occurs, we performed a manual inspection of Stack Overflow posts (Table 3.2). We inspected a random sample of 30 posts for each pair to keep categorization tractable, as done in Barik et al. [Bar18a]. We manually excluded posts that

[2] `https://go.ncsu.edu/cross-lang-study`
[3] `https://cloud.google.com/bigquery/`

Table 3.1 Participants interviewed

ID	Exp[1]	Domain	Recent Transition
P1	15	Compilers	C# ⇒ Python ⇒ C++
P2	9	Data Science	Python ⇒ Julia
P3	18	Information Sciences	Python ⇒ PHP
P4	15	Neuroscience	R ⇒ Python
P5	10	Security	C++ ⇒ TypeScript
P6	20	Cloud Services	C# ⇒ TypeScript
P7	6	Cloud Services	C# ⇒ Python
P8	10	Web Platform	C# ⇒ JavaScript
P9	31	Data Science	C# ⇒ JavaScript ⇒ Scala
P10	8	Business Applications	C# ⇒ Rust
P11	12	Web Platform	C# ⇒ Ruby
P12	10	Data Science	Python ⇒ SAS
P13	6	Software Engineering	C++ ⇒ JavaScript
P14	10	Data Science	R ⇒ Python
P15	20	Software Engineering	C# ⇒ Swift
P16	5	Data Science	R ⇒ Python

[1] Years of self-reported programming experience.

did not make any explicit connection between the languages of each pair, sampling another random post to replace it as necessary. Because the inclusion and exclusion criteria can have multiple interpretations, the first two coauthors labelled a random sample of 30 posts. This labelling had 100% agreement between the coauthors, and suggests a clear understanding of how to categorize posts. The two coauthors proceeded to label the rest of the Stack Overflow posts using the following classifications:

- *Correct:* The post makes a connection to a previous programming language with correct assumptions regarding the target language as revealed by the accepted answer, or
- *Incorrect:* The post makes a connection to a previous programming language with incorrect assumptions regarding the target language as revealed by the accepted answer.

Next, we calculated inter-rater reliability (IRR) between the two coauthors (Cohen's $\kappa = 0.89$), and obtained "substantial" agreement [Lan77]. We discussed disagreements on whether a post was correct or incorrect: if there was still disagreement, it was reconciled by the first author. Finally, we calculated the percentage of correct and incorrect posts. We used instances of correct and incorrect assumptions as evidence of cross-language interference and facilitation.

3.2.3 Phase II: Study Design for Interviews with Professional Programmers

To answer RQ2 and RQ3, we conducted semi-structured interviews with professional programmers.

3.2.3.0.1 Participants

We used *purposive sampling* [Ton07] to recruit 16 professional programmers who were learning a new programming language within the past 6 months (Table 4.1); these participants were still early in their learning process and working through their initial stumbling blocks in the new language. The participants (12 male, 4 female, self-reported) were from large software, technology, and data analytics companies with years of programming experience ranging from 5 to 31 years ($\mu = 12.8$, $sd = 6.6$). There were a total of 14 unique language transitions. Before the interview, participants completed a background questionnaire asking them about their previous languages and an obstacle they have experienced while adapting to the new language.

3.2.3.0.2 Protocol

We conducted semi-structured interviews either on-site or remotely, within 60 minute time blocks. Two of the authors conducted and recorded the interviews separately. All sessions were conducted with a single observer and a single programmer. We used the following structure for questions: 1) participant background, 2) first steps, 3) obstacles, 4) learning process, and 5) general strategies. The background information from the questionnaire was used to tailor the questions for the participants. The semi-structured interview format allowed the flexibility to ask questions impromptu and dig deeper into more specific obstacles. The recordings were later transcribed by the first author for analysis.

3.2.3.0.3 Analysis

RQ2: How do experienced programmers learn new languages? To answer RQ2, we conducted inductive thematic analysis [Bra19] on the interview transcripts over multiple phases: transcribing interviews, generating open codes by labelling notable recurring statements made by the participants, identifying relationships between the codes, and organizing them into meaningful themes.

RQ3: What do experienced programmers find confusing in new languages? To understand how programmers confuse language concepts, we selected themes from our analysis that highlighted interference due to previous programming knowledge.

3.3 Results

3.3.1 RQ1: Does cross-language interference occur?

Cross-language interference occurs on Stack Overflow across various language pairs. We found a total of 276 instances of incorrect assumptions (Table 3.2), which is around 61% of the 450 posts inspected. There were a total of 174 posts with correctly stated assumptions, which is only around 39% of the total posts. It's important to note that this provides evidence of interference occurring but does not imply programmers have incorrect assumptions 61% of the time. The `<Kotlin, Java>` pair

Table 3.2 Posts by Programming Language Pair

Language Pair[1]	Posts[2]	% Accepted[3]	Correct[4]	
			n	%
<C, C++>	30863	65%	9	30%
<C#, Visual Basic>	11522	62%	8	27%
<Objective-C, Swift>	9416	50%	10	33%
<Python, C++>	6763	51%	15	50%
<Java, C#>	6748	59%	16	53%
<Scala, Java>	6622	55%	8	27%
<PHP, Java>	6152	46%	16	53%
<R, Python>	2824	49%	12	40%
<Kotlin, Java>	2565	53%	6	20%
<Matlab, Python>	2407	53%	11	37%
<Node, PHP>	2077	40%	14	47%
<Ruby, Python>	1314	65%	14	47%
<Perl, Python>	1152	67%	13	43%
<Lua, C++>	1143	63%	12	40%
<Clojure, Java>	1098	68%	10	33%

[1] The pair of programming languages.
[2] Total number of questions where the two languages are tagged or referenced in body.
[3] Percentage of questions that have accepted answers.
[4] Total posts (out of 30) classified as having correct assumptions formed from prior language knowledge.

had the highest number of posts with incorrect assumptions, which reflects the Java programmer's confusion mentioned in **??**. The next two pairs, `<C#, Visual Basic>` and `<Scala, Java>`, also contained a high number of incorrect assumptions. However, there were other pairs like `<Python, C++>`, `<Java, C#>`, and `<PHP, Java>`, which had a more even distribution of posts with correct and incorrect assumptions; this suggests easier transitions between the languages. While reviewing the 450 Stack Overflow posts, we encountered instances where programming languages behaved in surprising ways for programmers. We highlight three examples, two of which involved interference between syntax and concepts, and one which involved facilitation—making it easier to use type inference.

Interference: R ⇒ Python[4]

s An R programmer is now using Python and its data processing library, Pandas. They are unable to successfully relate their previous knowledge about subsetting, in R, to Python: "I'm seriously confused. Maybe I'm thinking too much in R terms and can't wrap my head around what's going on in Python."

They present the R expression they want to translate, as well as several attempted translations in Python:

```
# R
data[data$x > value, y] <- 1
# Python
```

```
data['y'][data['x'] > value] = 1
```

Several concepts in R interfered, but we will highlight the most significant: Python prevents assignment to copies of dataframes. In this case, the indexing operation `data['y']` returns a copy of the dataframe and setting the value with `[data['y'] > value] = 1` will not work as the R programmer expects. The knowledge that the equivalent R expression will set the value of 1 without any warnings interferes with Python's warning.

Interference: PHP ⇒ JavaScript[5]

A PHP programmer who has switched to programming in JavaScript asks how to store transient information (sessions), such as application state about a user. Typically, PHP uses server-side session variables (`$_SESSION`) for this purpose. While related concepts, such as local storage and browser-based sessions exist, the programmer is warned that sessions cannot be safely and securely stored directly on the client—the programmer's knowledge about server-side sessions leads to a faulty assumption about their applicability in other programming contexts.

Facilitation: Java ⇒ Kotlin[6]

A Java developer is learning Kotlin. They ask if the following Kotlin expression can be simplified:

```
val boundsBuilder: LatLngBounds.Builder = LatLngBounds.Builder()
```

The developer suspects their declaration is more verbose than it should be, given their knowledge of local variable type inference in Java. They assume the declaration can be simplified:

```
val boundsBuilder = LatLngBounds.Builder()
```

This is an example of facilitation—the accepted answer confirms that the developer can simplify the expression because Kotlin supports type inference, allowing for the explicit type declaration to be removed.

These examples illustrate how previous knowledge of language syntax and concepts interact with knowledge learned in a new language. In some cases, this results in interference, which harms a programmer's ability to grasp new syntax and concepts in the new language. In other cases, this results in facilitation, which helps programmers make meaningful connections to previous languages and helps them learn the new language.

[4] https://stackoverflow.com/questions/30923882
[5] https://stackoverflow.com/questions/47137666
[6] https://stackoverflow.com/questions/38131655/

Cross-language interference occurs across various language transitions on Stack Overflow posts. We found that 61% of the 450 posts contained incorrect assumptions about the target language, and only 39% contained correct assumptions.

3.3.2 RQ2: How do experienced programmers learn new languages?

We present the themes on how experienced programmers learn new languages. A summary of the themes is listed in Table 3.3.

3.3.2.1 Programmers learned languages on their own

Programmers who switched teams lacked formal training for the new language and its associated technology stack, leaving learning to themselves. For example, when P1 switched from C# to Python for a new project, there wasn't any training involved and the on-boarding process was, "hey we want to get exposed to the Python world, go get started!" Although some programmers were given training initially on the project, "realistically for learning the new language [they] were pretty much on [their] own" (P7). This forced programmers to watch "language tutorial videos on Pluralsight"[7] (P5) or read online documentation. Some programmers "got initial tips from some folks from the team on what's what" (P6), and when running into complex issues "reached out to the group and said has somebody else hit this before?" (P1).

3.3.2.2 Just-in-time learning is a dominant strategy

To learn new languages, every programmer we interviewed used *just-in-time learning* [Bra09], an opportunistic strategy focused on only learning features as needed. Given time constraints, programmers made use of immediately available resources like online documentation, video tutorials, online searches, and available experts. Traditional resources like programming language books were only used as a reference, since programmers "just don't have time to do that" (P5). Programmers were primarily concerned with completing tasks in a reasonable time and "figuring out how to not burn tons of time on a single problem" (P1). Quicker resources, like cheat sheets, were preferred for language transitions. For example, the first thing P2 did was to make use of cheat sheets [Qua17] to help them transition from Python to Julia. P15 was also a fan of cheat sheets:

> *It seems like if you were going from one framework to another, from one technology stack to another—even if you're not going from A to B, you're just starting off on B—there's probably a content cheat sheet that every dev needs to know.* (P15)

3.3.2.3 Programmers related the new language to previous languages

To help accelerate the learning process, programmers generally tried to relate the new language to their previous languages. Programmers started by "loosely taking ideas from working in another

[7] `https://www.pluralsight.com`

language" (P14) or looking at existing code because "it's already probably been written and it's out there somewhere or at least something close to it" (P1). While this learning strategy was useful for bootstrapping, some programmers started from scratch. For example, when moving from C# to Ruby, P11 described "trying to be very conservative and mindful and trying not to map anything over, but just treating everything as something brand new." Similarly, P12 explained that they did not try to map things from Python when learning SAS "mostly because the syntax was so new that every time [they] tried to do anything, [they] would have to go and google the syntax." P10 expressed a similar problem when learning about managing memory in Rust after years of using C#: "there wasn't a clean way for me to just get there. I had to go and learn that stuff from scratch." These examples illustrate that programmers typically reuse knowledge—if possible—but sometimes avoid doing so when it's more troublesome.

Programmers use an opportunistic learning strategy, relating syntax and concepts of the new language with their previous language. This offers expediency but causes interference when major differences exist between the two languages.

3.3.3 RQ3: What do experienced programmers find confusing in new languages?

We present the themes explaining how programmers confuse language concepts. A summary of these themes is listed in Table 3.3.

3.3.3.1 Old habits die hard

Programmers had to constantly suppress old habits acquired from previous languages. For example, P3—who was used to Python—had trouble adapting to block delimiters in PHP, where "it's near-impossible to figure out exactly which opening brace you're closing once your HTML/PHP gets to any complexity at all." Similarly, P15 realized that "in Swift, the open curly bracket needs to be on the initial line of the method declaration and if you put it on the next line the method may not execute in an expected fashion." There were minor but frustrating difference like 0 versus 1 indexing for lists in languages such as Python and R. P4 described their frustration in "typing `a[1]` thinking that it's `a[0]`, and then wasting 5 minutes like a complete fool not understanding why nothing makes sense" (P4). Programmers are able to resolve these small differences, but it still causes interference at the onset of learning a new language.

3.3.3.2 Mindshifts are required when switching paradigms

Some language transitions required fundamental shifts in mindsets, or "mindshifts" [Arm07]. For example, when P2 transitioned from Python to Julia, they were constantly trying to make an object and realizing that "there's no objects, there's only structs!" With Julia, they needed to write more functional code, a shift from the object-oriented programming that they were used to in Python: "it was just needing to shift that and realize I'm never gonna write 'something-dot-something-else' ever

or rarely." For P10, they had to completely rethink the problems they would have normally solved in C# because of Rust's unique ownership feature for memory safety:

> *A really fascinating thing about learning Rust was that when I went and started to do these things—things that I would reach for in C# that I knew would work—Rust wouldn't allow it and as a result I had to rethink the problem and re-implement it in a way where the ownership characteristics of that algorithm were very explicit.* (P10)

Another big paradigm shift occurred for P5, P6 and P13—all transitioning from imperative or object-oriented coding to event-driven and asynchronous coding—forcing them to think differently. The programmers had to learn brand new concepts in JavaScript like asynchronous programming or "shadow and virtual DOMs" (P13). P6 described how it was difficult making sense of asynchronous code because "you got a whole bunch of '`async/await` mode' working in your mind and you have to convert it." To make matters worse, "the most confusing part is there are a couple of ways to do asynchronous programming, with observables or promises" (P13). For P5, whose background was in C++, the front-end coding in TypeScript was a big challenge because "for the back-end, the code I think is more straightforward. You have the logic and most likely you know single places you'll handle it. It's not like the UI" (P5). Here, the interference issues aren't due to any particular syntax or concept but the way one solves problems in the new language.

3.3.3.3 Learning a language is difficult when there is little to no mapping with previous languages

Programmers had a harder time learning the new language when there was little to no mapping of features to previous languages. For example, P12 could not make sense of some fundamental programming language features of SAS that were clear in Python, like statements versus method parameters. They could not understand "why some things are statements that affect a procedure, but aren't parameters" and were "still confused about the overall syntax and what is or isn't a statement"—even after having worked in the language for a few weeks. A drastic example was P5, who experienced a big transition from C++ to TypeScript, resulting in *tech shock*: "Everything is different! Not just the programming language—the IDE, source control, everything is different." P13, who underwent a similar language transition, found that concepts were challenging in JavaScript because they "could not equate it back to C++." Due to limited mapping of features to previous languages, programmers could not make full use of facilitation to learn the new language.

In the extreme case, programmers were forced to learn a completely foreign syntax or concept, in particular, when it was an essential built-in feature of the new language. For example, P9 had difficulty learning traits in Scala because they "never had a language with traits before. Traits have a default implementation and understanding what would be performant and what wouldn't—and when to use what—that was the tricky part." P7 learned that for Python, "the major difference is the multiple inheritance thing, that Python inherits from the C++ world, which supports multiple inheritance. In C# you can't do that." In another case, the difficulty was due to differences in memory

management, for example, when P10—who previously used C#—was learning Rust:

> *There's a very alien concept in Rust that is the borrow checker, which is the concept of having the compiler verify more things, and the way it does it is somewhat esoteric. That's very alien, and that's something that I think is really cool but it's also very rough at the moment and so that's kind of something that's been the biggest struggle when trying to learn Rust.* (P10)

Even within the same context, such as data analysis or mobile applications, the lack of mapping caused a lot of confusion. For example, P14, who switched from one data analysis language (R) to another (Python/Pandas), could not find an immediate equivalent for R's `spread` and `gather` functions: "Pandas already had the functionality but it was more hidden using drop level and unstack. These were really hard to understand in Pandas—it was some pretty weird stuff." Similarly, P15, who switched from C# to Swift, was very surprised to learn how the user interface code and its graphical layout view in Xcode were connected: "Knowing that you can't interact with a UI object straight out of the box from the code is very important. Once you draw the referencing outlet connection between View and Controller you can trigger methods and `get/set` properties as you'd expect in the .NET world."

3.3.3.4 Searching for the right terminology and code examples is difficult

We found that moving to a new programming language presents a *selection barrier* [Ko04b], making it difficult to search for information about the language and its associated technologies. Programmers recounted trouble acquiring the vocabulary even before performing the search. For P12, the names for the same structures in Python/Pandas were slightly different than SAS where a "dataframe is data set, a row is an observation, a column is a variable." When they tried to plot with SAS, they "don't know what the name of the `proc` for plotting in SAS is so [they] have to start looking that up first, then find documentation for a couple different ones, then have to figure out how to make them work." On the one hand, "it's the breadth of the libraries that usually get you, you don't even know what exists, what to even look for to see if something is already there" (P1). On the other hand, insufficient search results provided little to no facilitation. For example, P4 had difficulty searching information for a Python library called seaborn—compared to the equivalent R library ggplot—because "it is just less documented. For ggplot, if you google anything, you get like 100 hits, and the top ones are bound to be good due to Google selection of results. With seaborn, you get like 10 hits."

Even when programmers found documentation and code examples, they were either incomplete or lacking in detail. For example, P8 expressed a frustration regarding testing libraries in JavaScript because "they have their own convention, TypeScript has its own convention, JavaScript has its own convention, it is actually mixing everything!" This was especially problematic when conventions found online weren't always the same ones used by the specific team: "There's a lot of conventions around the language. In C++, the styles can change a bunch from team to team" (P1). For some

languages, the documentation was either lacking in quality or was completely missing. For example, P2 was frustrated with the Julia documentation because "it was so useless for figuring out the imports." Similarly, P12 expressed that the SAS documentation "only tells you how to copy-paste and run a simple program, leaving you completely mystified as to how the execution and control flow of a SAS program works." This lack of depth can lead to frustrating experiences for programmers when they had better documentation in previous languages, such as P15: "Xcode documentation samples were pretty good enough to where they would run. But the documentation, MSDN, and the available samples for creating Microsoft platform-based applications were tenfold deeper and richer and easier for to use."

3.3.3.5 Retooling is a necessary and challenging first step

Finally, before programming in the new language, programmers faced difficulty retooling themselves in a new environment. This typically involved adapting to the discrepancies of the new integrated development environment (IDE) for programming in the language. Although programmers were able to adapt to basic features of IDEs (facilitation), there was interference when some aspects of the IDE differed from their previous IDEs. For example, P15 discovered that in Xcode "build targets aren't 'Universal' in definition (like .NET) and when terminologies are shared across platforms but don't implement the same notion, you're lost for days!" Interestingly, for P9 there was interference when they tried building their Scala project in IntelliJ because the IDE attempted to support Scala, but continued presenting dialogs in the previous language:

> *Part of the problem is IntelliJ is aimed at the Java developer and I'm using SBT, which is from the Scala world. And it's sort of importing the SBT into the concepts in the IDE of IntelliJ. So I'm looking at dialogs that are all about Java and which JDK and that doesn't map to what I wrote in the declarative SBT language.* (P9)

Other concerns regarded either a lack of IDE features or learning new features that were distracting. P2 had been "spoiled with Python and PyCharm" and found it very difficult to find proper IDE support for Julia; they just wanted "an IDE that does syntax highlighting and IntelliSense-like autocompletion." P1 found that learning a new feature—like debuggers—effectively halts a programmer's progress on actual tasks and are distracting "because you're learning and debugging at the same time as opposed to just debugging once you're fluent."

However, sometimes the transition to new tools in the language also benefited programmers. For example, P5 found it a lot easier moving from MSBuild (C++) to Gulp (JavaScript), which allowed fast build cycles when developing TypeScript applications. In particular, the DevOps pipeline helped them make progress much quicker:

> *I think right now the build system for us, I think it's better since now we are using DevOps—a pipeline to build the code. It's very easy for us to even schedule the private build and also it's very easy for us to quickly get new things, check in the code, test it, and even build things on top of it.* (P5)

Programmers confuse a new language's syntax and concepts with previous languages, leading to a number of issues like trying to suppress old habits, wrestling with mapping issues, struggling to find and use proper documentation, retooling and shifting one's mindset for new paradigms.

3.4 Limitations

Our mixed-methods approach of investigating Stack Overflow and conducting interviews introduces certain trade-offs and limitations.

The choice of sampling technique in our Stack Overflow analysis has several trade-offs [Mos52]. Because the sampling approach is non-probabilistic, it does not allow for sample-to-population, or *statistical generalization*. Rather, our approach targets diversity (rather than representativeness) in order to identify evidence of interference across many different programming languages.

We used correct and incorrect assumptions as a proxy construct for facilitation and interference. While this approach provides us with a useful, high-level characterization of the Stack Overflow posts, there are potentially additional insights that we could learn had we performed a more intricate qualitative coding technique, such as open coding. The trade-off for doing so is that open coding is significantly more costly to execute. Instead, we conducted semi-structured interviews with experienced programmers to delve deeper into cross-language interference.

The posts we examined on Stack Overflow as well as our interviews do not completely cover the set of all language transitions, as the full permutation space of language transitions is intractable. Our approach attempts to cover language transitions that are most likely to occur in practice. Consequently, there may be some interference issues that our study was not able to identify.

Finally, we acknowledge that qualitative research, however rigorously conducted, involves not only the qualitative data under investigation but also a level of subjectivity and interpretation on the part of the researcher as they frame and synthesize the results of their inquiry. To support interpretive validity, we followed the guidelines set by Carlson [Car10] and performed a single-event member check with our results. Six participants who replied agreed with our presentation of the results and only wanted minor changes to their quotations. Additionally, we emphasize that interference theory is only one of many possible lenses through which we can organize and present our findings. Other theories, such as *notional machines*, have also been used to identify and explain programming conceptions [DB86; Bou81; Ber14].

3.5 Related Work

3.5.0.0.1 Novice misconceptions

Programmers often have misconceptions while learning new programming languages, but most studies have focused on novices. Swidan et al. [Swi18] proposes "intervention methods to counter those misconceptions as early as possible," but this work is primarily targeted to novices. Similarly,

Kaczmarczyk et al. [Kac10] has examined misconceptions and how to measure them for novices. In contrast, the novelty of our work is towards experienced programmers who need to switch languages [Mey13], and requires methods of learning distinct from those designed for novices [Swe03; Kel05; Guo13]. Our study investigated switching languages for experienced programmers and took the first steps in examining how knowledge of previous languages can interfere.

3.5.0.0.2 Programming language transitions

There are a few studies on transitions between programming languages. Scholtz and Wiedenbeck [Sch90] studied experienced Pascal or C programmers writing a program in a new language, Icon, and found that they were strongly influenced by their knowledge of what would be appropriate in previous languages. Wu & Anderson [Wu90] conducted a similar study where programmers who had experience in Lisp, Pascal and Prolog wrote solutions to programming problems and found that solutions written in one language facilitated learning in another language. Uesbeck & Stefik [Ues19] studied the effect of using multiple languages in a controlled study, where participants implemented several variations of database queries: some variants involving the same language, while others mixing SQL and Java. While the results were inconclusive, the authors suggest that the methodology could be effective for studying the productivity costs associated with mixing languages. We examined empirical evidence and conducted interviews to understand the transition from one language to the next for various contexts. We also investigated how programmers confuse two different languages using the lens of interference theory [Und57].

There have been fewer studies on interventions for learning new languages. Bower & McIver [Bow11] explored a new teaching approach called Continual And Explicit Comparison (CAEC) to teach Java, using facilitation, to students who have knowledge of C++. They found that students benefited from the continual comparison of C++ concepts to Java. Shrestha et al. [Shr18] used a similar technique using a tool called Transfer Tutor to teach R from the perspective of Python; programmers who used the tool found the comparisons between the languages useful. These intervention techniques might benefit programmers who learn new languages from the perspective of a known neighboring language, but there are a number studies on larger transitions—for example, from procedural or imperative to object-oriented languages [Dét95; Nel97; Nel09; Arm07]. These studies have shown professional programmers experience greater interference as they have to make fundamental shifts or "mindshifts," which might require further support. In this study, we have uncovered interference issues in the modern context and examined numerous language transitions. We also found other issues that have not been explored like dealing with little to no mapping of language features (Section 3.3.3.3) and retooling (Section 3.3.3.5), which have implications for future tools and techniques.

3.5.0.0.3 Programming knowledge

Knowledge structures have been proposed for how programmers encode semantic [Shn79] and domain information [Bro83] about a program as well as *prime structures* [Lin79], that include elements of syntax, control-flow and data-flow [Pen87] of the program. These knowledge structures [Ric81] have been formalized and referred to as *programming plans*. Programming plans act like schemas that are first instantiated and then its slots are filled with concrete values as a programmer builds an understanding of the code [Sol82]. Plans may help programmers fill in the "gaps" when trying to understand code.

Gilmore & Green [Gil88] suggested that programming plans may not generalize across different languages, and that plans cannot represent the underlying deep structure of programs. Bellamy & Gilmore [Bel90] examined the protocols generated from experts in different languages as they created programs. Using two different models of programming plans, they found neither model was well supported by protocols; further, different programming language experts generated different types of representations. We believe our results provide further insight as to why plans may not generalize across languages. For example, we found programmers tend to relate a new language to previous languages (Section 3.3.2.3), which suggests an attempt to reuse previous programming plans as a bootstrapping strategy. However, due to interference issues, the previous plans might either need significant modifications (Section 3.3.3.3) or be replaced entirely (Section 3.3.3.2), depending on how closely related the two languages are.

3.6 Discussion and Design Implications

Our findings demonstrate that interference is not an isolated phenomenon; indeed, in Stack Overflow, instances of interference are found across all of the programming languages we investigated. Furthermore, in our interviews, participants reported that interference arises routinely as they learn a new language—for example, from having to suppress old habits from previous languages (Section 3.3.3.1) or having to "rethink the program" (P10) due to a substantially different paradigm (Section 3.3.3.2 and Section 3.3.3.3).

As opposed to traditional classroom environments where one learns "step-by-step" (P5), experienced programmers in our study used opportunistic strategies to learn essentially "on [their] own" (P7) or "learning through work" (P13), for example, using online resources or asking teammates (Section 3.3.2.2) [Bra09]. Unfortunately, these informal approaches to learning sometimes result in an incomplete lens for how the language works, resulting in "unintentional bugs" (P5) and other difficult-to-diagnose problems in the code when something doesn't work as expected.

In the remainder of this section, we present design implications for technical authors, toolsmiths, and programming language designers that can help reduce some of these interference difficulties for programmers.

Implication I—Design documentation that reduces interference and supports knowledge trans-

fer. Programmers in our study desired more accessible resources that leveraged the programming knowledge they already have (Section 3.3.2.2 and Section 3.3.2.3). Such resources included "cheat sheets," which present code snippets that map their familiar language to their new language (P2) and relate concepts they already know "from working in another language" (P14), to the new language tutorials, and even resorting to "reading other people's code" (P3, P15) to understand the programming language idioms.

Our findings suggest that resources that teach languages through relating a new language to a known language are more useful and accessible to programmers than resources that present the new programming language in isolation. Several books [Dac96; Jon02; Ohr17; Zha], blogs [Blo; Clo], language documentation [Rub; Pan; Wil18], and training courses [Dat; Mat] embody this pedagogical strategy.

However, these resources—while useful—are essentially hand-crafted through the authors' intuitions about what misconceptions the programmer might have, and not necessarily the ones that programmers actually have. While misconceptions about novice programmers are readily found in the literature [Kac10; Qia17; Dan12], misconceptions experienced programmers have are comparatively understudied. Shrestha & Parnin [Shr19] presented three possible instrument designs which can be used for discovering and validating misconceptions when switching languages for experienced programmers. Such research is needed to make learning resources more effective and relevant to experienced programmers.

Implication II—Build automated tools to provide on-demand feedback. Although technical documentation is useful, these resources are decoupled from where the programmer needs the most help—in their program environment as they work (Section 3.3.2.2 and Section 3.3.3.4).

Automated tools can help with this. For example, Johnson et al. [Joh15] propose "bespoke" notification tools that provide adaptive feedback to the programmer based on the programmer's prior knowledge of programming languages and concepts. Python 3 adopts this idea of using prior programmer knowledge to assist programmers who come from a Python 2 background, through hard-coded error messages: in Python 2, `print` does not require surrounding parentheses, while in Python 3, `print` is a function and thus must be called like any other function:

```
>>> print "Hello"
  File "<stdin>", li
    print "Hello"
                ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
 ↪  print("Hello")?
```

The `SyntaxError` message makes the assumption that this error is due to a misconception (or ingrained behavior) instilled from experience with Python 2. We can repurpose this idea generally to language transitions and help programmers more efficiently resolve error messages that they

might otherwise only "eventually figure out" (P1) after spending substantial time and effort.

Implication III—Be intentional about programming language syntax, semantics, and pragmatics. Certain programming languages anticipate that new adopters arrive through common pathways. That is, we expect most new Rust users to come from systems programming languages like C++, and we expect most new TypeScript users to come directly from JavaScript. For these users, intentionally designing language features by considering interference effects can reduce barriers (Section 3.3.3.2 and Section 3.3.3.3) to adopting the new programming language.

As an example, a substantial barrier to new Rust users is the borrow checker—a compile-time feature that helps enforce safe memory management [Zen19]—which our own participants described as "a very alien concept" (P10). Even the Rust manual concedes that borrow checking has a costly "learning curve" and that programmers "fight with the borrow checker" because their "mental model of how ownership should work doesn't match the actual rules that Rust implements" [Rus]. Interference theory also explains these difficulties: for some programmers, the borrow checker is so unfamiliar as a concept that they have no prior support to *facilitate* learning; and for other programmers, borrow checker concepts at a casual glance seem similar to existing models, such as "resource acquisition is initialization" (RAII), in C++, but ultimately functions differently enough that it *interferes* with their past knowledge.

Intentionally considering these adoption pathways as part of language design can reduce these interference challenges. For instance, the "primary goal of TypeScript is to give a statically typed experience to JavaScript development" and "the intention is that TypeScript provides a smooth transition for JavaScript programmers—well-established JavaScript programming idioms are supported without any major rewriting or annotations" [Bie14]. But providing this smooth transition has a costly consequence: "the TypeScript type system is not statically sound by design."

As the two examples illustrate, designing for interference requires making difficult design tradeoffs. But if we want to design programming languages that people actually use, we need to consider how our language design decisions interfere or facilitate with our anticipated programmers' prior knowledge.

Implication IV—Support not only programming languages, but programming language ecosystems. Issues with interference when learning new programming languages are exasperated when new programming languages bring with them new programming language *ecosystems*—that is, "everything is different, not just the programming language" (P5), but the environment in which the programmer builds, edits, debugs and tests their code (for example, *tech shock*, Section 3.3.3.5).

To address these challenges, React developers provide tool support to welcome programmers into the new ecosystem. Specifically, the `create-react-app` [Cre] is an integrated toolchain that abstracts away the complexities of third-party library management, live-editing, optimization, and configuration. `create-react-app` allows the user to quickly and easily begin experimenting with the library until the programmer is comfortable enough to *eject* from the `create-react-app`

toolchain.

A second method to minimize interference issues from ecosystems is to unify the underlying tooling environment, or at least provide the programmers with a unified tooling experience. From this perspective, we would recommend that toolsmiths and language designers add support for programming languages to well-established integrated development environments, rather than providing custom tool and editing experiences. For instance, the language server protocol (LSP) [Lan] allows programming language support to be implemented and distributed independently of any given editor or IDE, as long as that IDE implements LSP.

In short, language designers should collaborate with tool designers so that programmers can more easily adopt new programming languages through editing environments that are already familiar to them.

3.7 Conclusion

We conducted a mixed-methods study to understand what impact previous programming language experience has on programmers in Stack Overflow questions across 18 different programming languages and semi-structured interviews with 16 professional programmers. From Stack Overflow, we found 276 instances of interference that occur across multiple languages. We then interviewed programmers who reported various challenges learning a new language like mixing up the syntax and concepts with their previous programming languages due to interference. We discussed design implications for technical authors, toolsmiths, and language designers, such as designing documentation and building automated tools that reduce interference, and welcoming programmers not just into a language, but its entire ecosystem. To answer the question posed in the prelude, even professional programmers have difficulties with learning programming languages, and we should offer tools and techniques to help them learn more efficiently and effectively.

Table 3.3 Learning Strategies and Language Interference Themes

Learning Strategies			
THEME	DESCRIPTION	REPRESENTATIVE EXAMPLES	PARTICIPANTS[1]
Learning on their own (Section 3.3.2.1)	Programmers lacked formal training for the new language and its associated technology stack, leaving learning to themselves.	"We didn't have a procedure for people getting up and running." "'I just do everything ad-hoc!" "I got initial tips from some folks from the team on what's what."	P1, P2, P5, P6, P7, P13, P14, P15, P16
Just-in-time learning (Section 3.3.2.2)	Programmers focused on only learning features as needed.	"There's probably like a content cheat sheet." "I didn't learn typescript step-by-step." "Step one for me is always find and read other people's code."	P1, P2, P3, P5, P9, P14, P15
Relating new language to previous languages (Section 3.3.2.3)	Programmers tried to map features of the new language to their previous languages.	"I loosely [take] ideas from working in another language." "I would try to find the counterpart of C++ in React." "If you can compare them side by side and find their similarities you're more than halfway there."	P1, P2, P9, P12, P13, P14, P15
Language Interference			
THEME	DESCRIPTION	REPRESENTATIVE EXAMPLES	PARTICIPANTS[2]
Old habits die hard (Section 3.3.3.1)	Programmers had to constantly suppress old habits from previous languages.	"I'm typing `a[1]` thinking that it's `a[0]`." "I still type the type first before the variable." "I'm gonna make it an object for this, no don't do that!"	P2, P3, P4, P6, P9, P15
Mindshifts when switching paradigms (Section 3.3.3.2)	Sometimes programmers wrestled with larger differences that required fundamental shifts in mindsets, or "mindshifts."	"All my assumptions were thrown out the window." "I had to rethink the problem and re-implement it." "There are lots of events and promises all these things makes it really hard to debug."	P2, P5, P6, P9, P10, P13, P15
Little to no mapping with previous languages (Section 3.3.3.3)	Programmers had a harder time learning the new language when there was little to no mapping of features to previous languages.	"There's a very alien concept in Rust that is the borrow checker." "I've never had a language with traits before." "I did not work with concepts like virtual DOM, shadow DOM before."	P2, P5, P9, P10, P11, P15
Searching for terms and documentation is hard (Section 3.3.3.4)	Programmers found it difficult to search for information about the language and its associated technologies.	"You don't even know what exists, what to even look for." "Scala is not that common. Some of it required a little deeper digging." "They have their own convention, TypeScript has its own convention, JavaScript has its own convention."	P1, P2, P4, P8, P9, P11, P12
Retooling is a challenging first step (Section 3.3.3.5)	Programmers faced difficulty retooling themselves in the environment of the new language.	"I was using Visual Studio to debug C# code and now it's gdb to debug C++ code." "In Xcode, build targets aren't 'Universal' in definition like .NET." "The problem is IntelliJ is aimed at the Java developer and I'm using SBT which is from the Scala world."	P1, P2, P9, P12, P15

[1] Participants who used a similar learning strategy.
[2] Participants who experienced the particular language interference theme.

CHAPTER

4

AN ONLINE COMMUNITY OF PRACTICE FOR DATA SCIENTISTS

Data science practitioners face the challenge of continually honing their skills such as data wrangling and visualization. As data scientists seek online spaces to network, learn and share resources with one another, each individual has to employ their own ad-hoc strategy to practice their data science skills. Given these disjointed efforts, it is crucial to ask: how can we build an inclusive, welcoming online community of practice that unites data scientists in their collective efforts to become experts? In this chapter, we discuss a study conducted on #TidyTuesday—a daily hashtag project for data scientists using R—as one solution to this problem.

4.1 Motivation

Data scientists are increasingly prevalent in online spaces. They are a group of people who are distributed across various parts of the world with diverse backgrounds, from statistics to bioinformatics to graphics. Data scientists also differ from traditional programmers and are typically end-users without a formal background in programming [Kan12]. To build up their expertise, they are faced with the constant challenge of practicing skills like acquiring, cleaning, wrangling, visualizing, and presenting data. To expedite their learning process, data scientists are becoming dependent on faster, more accessible resources which are typically found online like tutorials, documentation, or Q&A sites [Vas14b; Vas14a]. However, data scientists can get socially isolated in their efforts for practice without a community of practice, which can negatively impact motivation for consistent practice. Without a community to grow in, data scientists also miss out on tacit knowledge like best

practices and techniques not captured in online resources. As data scientists seek help in online spaces, it is crucial to ask: how can we build an inclusive, welcoming online community of practice that unites data scientists all over the world in their collective efforts in becoming experts?

Daily hashtags have been used by several online communities on Twitter to organize discussions around a topic, activity, or event. Twitter hashtags (#) have been previously used for trending "tweet chats" around activism such as challenging engineering stereotypes [Liu17] (#ILookLikeAnEngineer), or exchanging knowledge during breaking news such as pandemics [Kos14] (#SwineFlu). A daily hashtag is a different type of hashtag which is used periodically. For instance, #AdventOfCode is a popular daily hashtag where in the month of December, each day presents a new programming puzzle. Programmers then post a tweet and share their thoughts about their own approaches in solving the puzzles. Daily hashtags can help build a community of practice (CoP) by allowing programmers of all skill levels to practice solving programming puzzles and network or exchange knowledge on Twitter. But, can daily hashtags provide an online CoP for data scientists? Thus, our research questions for this study are:

- **RQ1**: Who participates in Tidy Tuesday and what are their motivations and goals?
- **RQ2**: What do participants gain by participating in Tidy Tuesday?
- **RQ3**: How does social activity around Tidy Tuesday cultivate a community of practice?

To investigate our research questions, we conducted a qualitative case study on #TidyTuesday—a daily hashtag project for data scientists to practice their data wrangling and visualization skills using R. #TidyTuesday provides data scientists access to a curated dataset in a GitHub repository every Tuesday. Participants perform their analysis of the dataset and produce plots answering exploratory questions of their own. They are encouraged to share a tweet with the hashtag including a link to their code and the plot they produced. #TidyTuesday can be characterized using the three main components of a CoP: the domain (data science), the people (data scientists), and the practice (data analysis and visualization). To understand the motivations and goals of #TidyTuesday participants and the social interactions that help form and sustain an online CoP, we conducted semi-structured interviews with 26 data scientists. The participants were from diverse backgrounds with varying skill levels from beginners to veterans, some of whom are widely known in the larger data science community. These characteristics provided us with the opportunity to explore a broad range of experiences which provide insights into why data scientists use #TidyTuesday, what the benefits are, and how it is used to cultivate an online CoP using [Wen02b].

From our qualitative analysis of #TidyTuesday, we found several motivations behind participation, and the ways in which the project successfully grows an online community of practice, which both corroborate previous findings and extend them. Participants' main motivation was to hone their data science skills with the help of weekly-released, curated datasets and a community of practice. #TidyTuesday participants underwent transformative experiences such as discovering numerous R packages and tools from others, improving data wrangling and visualization skills,

building data visualization portfolios for the job market, and supporting offline events like workshops and "hacky hours". We discuss how #TidyTuesday enabled these experiences by relating the project to constructs of a CoP and its design components [Wen02b] such as providing a rhythm, and having a loose and flexible structure to fit each individual's needs. However, we also identified barriers of entry for newcomers such as not knowing how to start and a general lack of constructive feedback and mentorship. To our knowledge, this is the first paper to explore the R community in cultivating an online CoP through the use of daily hashtags on Twitter. The key contributions of this paper are:

- The first qualitative study of #TidyTuesday, a daily hashtag that formed an online CoP for data scientists using R, through semi-structured interviews with 26 data scientists.
- An analysis of the intrinsic and extrinsic motivations behind participation in #TidyTuesday, the benefits gained by participants, and the social interactions that helped grow and sustain the project.
- A discussion of the design trade-offs of using daily hashtags on Twitter and a set of guidelines to successfully grow and sustain an online CoP for data scientists, and overcome learning and social barriers.

4.2 Related Work

In the following subsections, we present findings in the CSCW and HCI literature with regards to community of practice, data scientists and the R community, as well as daily hashtags on Twitter. We highlight the gaps in knowledge with how to foster an online community of practice through the use of Twitter for data scientists.

4.2.1 Communities of Practice

Communities of practice (CoP) are groups of people who share a concern, a set of problems, or a passion about a topic, and who deepen their knowledge and expertise in this area by interacting on an ongoing basis [Wen02a]. We study a nascent **R community** forming on Twitter around the #TidyTuesday project, designed to provide an online CoP for **data scientists** to **practice data wrangling and visualization**. The relevant literature we examined for studying #TidyTuesday include the design components necessary for cultivating a CoP developed by Wenger et al. [Wen02b], the idea of Twitter as an imagined community [And06] by Gruzd et al. [Gru11], and sense of community (SoC) theory by McMillan & Chavis [McM86]. Wenger et al. [Wen02b] describe how a CoP is different from organizational design which focuses on fixed goals and elements, where optimizing for *aliveness* is emphasized because a community has to invite interactions to keep it alive and growing. The authors suggest creating rhythm for the community, which daily hashtags like #TidyTuesday is designed to provide. Gruzd et al. [Gru11]'s study of a single member's Twitter network is also important because they found that Twitter can meet Jones [Jon97]'s minimum requirements for a virtual

settlement like interactivity, or sustained membership over time. The authors found that a single individual can form their own personal community on Twitter through network analysis, while we study many individuals who are joining a growing community of practice around #TidyTuesday using a qualitative approach to gain rich insights into motivations, and social interactions that help cultivate an online CoP. Finally, McMillan & Chavis [McM86]'s "Sense of Community" (SoC) theory outlining characteristics of a community like the fulfillment of needs is also relevant to how well #TidyTuesday meets the R community needs. We extend the literature on online CoPs by applying the framework on the #TidyTuesday project on Twitter and contribute new perspectives on how a data science CoP can be formed and sustained.

The CSCW and HCI communities have used the CoP framework as a lens for studying various groups in social media sites. For example, Marlow & Dabbish [Mar14] studied graphic designers and the social transparency provided by SNS (Social Network Site) features of a design portfolio website called Dribbble [Dri]. They found that SNS functionalities like following members and having access to artifacts supported social learning via *legitimate peripheral participation* (LPP) [Lav91] and professional identity development. Holikatti et al. [Hol19] also found heavy use of LPP in Facebook groups for learning how to host living spaces using AirBnB [Air], a sharing economy platform for hosting living spaces all over the world. They found that members learned affordances of AirBnB through Facebook by asking questions and interacting with more experienced AirBnB users. In our study, #TidyTuesday on Twitter facilitates these social transparencies via public tweets and links to code, which provides opportunities for skill and professional development via LPP. Kou et al. [Kou18] used CoP as a framework to study the changing practices of user experience (UX) professionals on reddit where they identified social roles in relation to knowledge production and dissemination in the online community of *volatile practice*—rapidly changing occupations. Data scientists face similar challenges in a young field that is ever evolving, and we provide an instance of using an online community of practice for those who are entering the field (more in Section 4.2.3). We extend the literature by studying how the R community use Twitter to improve their data wrangling and visualization skills, learn from each other, and gain a sense of community in an evolving field.

There is also prior work which examines accessibility, motivations and barriers in various online CoPs. For example, Mugar et al. [Mug14] studied the accessibility of participation norms in online communities where participants lack full access to others' work. The authors combined the theory of legitimate peripheral participation with the theory of social translucence to derive practice proxies such as traces of user participation in online environments that act as resources to orient newcomers towards the norms of practice. Similarly, Xu & Bailey [Xu12] uncovered motives for participation and expectations of the critiques within an online community. They provide recommendations for improving the design of systems that support community-based critique of creative artifacts. Our study provides similar insights within the data science context with regards to motivations behind participation in a community-led project like #TidyTuesday, discussing how Twitter can facilitate LPP, but also limit distributed critique.

4.2.2 Data Scientists

People across a wide range of professions now write code as part of their jobs with the purpose of obtaining insights from data rather than building software. The popular term for this type of work is "data science" and the group of people are often called "data scientists". Data scientists come from various backgrounds like engineering, business, design, and research [Dav12]. They are increasingly prevalent in both industry and academic settings. In industry, data scientists work in numerous sectors like public policy, technology, and healthcare [Loh17]. In academia, data scientists are graduate students, professors and technical staff writing code to make research discoveries [Guo12]. Data scientists are known to be like end-user programmers, writing code as a means to an end—to gain insight into data. Unlike traditional programmers, they are also a group that heavily engages in exploratory programming [Ker17c]. In particular, data scientists heavily engage in exploratory data analysis where they continually explore questions about the data and iteratively refine statistical models and visualizations to paint a story. However, data scientists also share similarities with software engineers, writing reusable analysis code to share with others—they engage in what Ko et al. [Ko11] call *end-user software engineering*.

Despite the growth of data science, there is little understanding of how data scientists are learning and practicing their skills outside of corporate and organizational settings. Prior work has given insight on the activities data scientists engage in at work [Guo12; Kan12; Ker17c; Kim16; Rul18; Seg07], and how practitioners teach beginners in both industry and academia [Kro19]. However, there are only a few studies examining how data scientists hone several skills such as acquiring, cleaning, wrangling, visualizing, and presenting data [Har15; Kan12]. To gain expertise in these skills, data scientists must decide between many different learning paths: attending a university and acquiring a data science major [Tre17; Van18], taking MOOCs (massive open online courses), or participating in hands-on workshops [Wil06] and coding bootcamps [Top; Cam20]. Despite data science programs becoming increasingly available, there is still debate around what to include in a data science curriculum [Bar18b] and how to prepare students for the industry.

Prior CSCW and HCI research has explored how data scientists work in both organizational and corporate settings, as well as in informal settings. There are several studies on solitary data science practices related to data wrangling tools [Guo11; Sut18], as well as exploratory analysis and barriers in computational notebooks [Ker18; Cha20]. In collaborative settings, Passi & Jackson [Pas17] and Passi & Jackson [Pas18] have found that data scientists collaborate in order to resolve tensions around trustworthiness of data and the analysis process. There has also been several studies on how data scientists collectively curate data [Mul19; Zag16], work together on a project [Zha20a; Wan19], share code in competitions [Tau17], and do data science for social good [Zeg18]. For example, Hou & Wang [Hou17] studied collaboration in two civic data hackathons, where data science workers help non-profit organizations and discovered unique broker roles. Tausczik & Wang [Tau17]'s study on Kaggle competitions found that data scientists re-use and share code with one another, which helped individuals practice and hone their skills. We extend these studies by providing insights about how a daily hashtag like #TidyTuesday helps data scientists hone their individual skills, while cultivating a

community of practice in the wild and outside of organizational, corporate, and hackathon settings.

4.2.3 The R Community

The R project [Rpr] was born in 1993 as a free and open source programming language and software environment for statistical computing, bioinformatics, and graphics [Iha96]. The R community is an open source community made up of an R-core, a team of software developers that maintain and evolve the R language, language users and package developers. The R community has also been the subject of extensive research in community evolution [Ger13; Vas14a] and the interplay between different channels [Vas14b] for asking questions such as Stack Overflow and mailing lists, where members are active in both channels but noticeable shifts towards the former in recent years. Zagalsky et al. [Zag16]'s study on the R community on Stack Overflow versus R-help mailing lists is especially relevant to our study. They describe how users exchange different types of knowledge on Stack Overflow and mailing lists, including a description of the reasons why members choose one channel over the other: users preferred Stack Overflow for the ability to gain peer recognition and faster turnaround on questions, while others preferred the R-help mailing list for its flexibility on topics and the high activity of experienced users.

There are several key players in the R community that have shaped the modern R community, making significant strides into promoting inclusivity, open source software, and education. RStudio [Rstb], a company behind the popular RStudio IDE (Integrated Development Environment) [Rsta] has been developing programming tools in R, making them more accessible to data scientists. RStudio is also a Certified B Corporation [Bco] company dedicated to creating and sustaining open source software for data science. Several leaders within the R community work at RStudio like Chief Scientist and Educator, Hadley Wickham and Software Engineer, Jenny Bryan, who have been pushing for more inclusiveness and diversity in the R community. Another key player is the R-Ladies Global [Rla], a worldwide organization whose mission is to achieve proportionate representation by encouraging, inspiring, and empowering people of genders currently underrepresented in the R community. They have over 138 chapters in 44 countries and 39000 members, holding meetups and events worldwide in order to introduce minority populations to programming in R. The Carpentries [Car] organization have a similar mission to foster diversity and inclusion as well as provide essential data and computational skills for conducting efficient, open, and reproducible research. The Carpentries hold workshops, develop openly available lessons designed using evidence-based teaching practices.

The R community has been shifting towards an online community on Twitter (#rstats) and undergoing a trend towards a new style of R programming called tidy R, which offer a user friendly and consistent way of doing data analysis and visualization. The #rstats community is prevalent on Twitter, and there is even an online textbook called "Twitter for R programmers" [Bar20] to help onboard non-Twitter users. Along with the move to sites like Twitter, there is also a popular programming paradigm in recent years called tidy R, comprised of packages in the "tidyverse" [Tida] that "share an underlying design philosophy, grammar, and data structures" of data [Wic14]—a

framework to tidy data and make it amenable for further analysis. #TidyTuesday encourages the use of this framework and provides yet another online resource to existing online channels like Stack Overflow and R-help mailing lists, but takes place on Twitter, where new types of knowledge might be created. The R community and its dynamics in social media sites like Twitter has not received attention of researchers, and we believe are one of the first to explore how Twitter can be used to share knowledge and learn from each other through #TidyTuesday.

4.2.4 Hashtag Movements on Twitter

Twitter has been extensively studied to explore how they are used by online communities to organize discussions around a topic, activity, or event. On Twitter, a hashtag (#) character is used for trending "tweet chats" around activism such as challenging engineering stereotypes [Liu17], or exchanging knowledge during breaking events [Cui12] such as natural disasters [Pot11]. There is an emerging body of research that is related to the online communities emerging on Twitter viewed through various analytic lens like social translucence, imagined communities, networks, or linguistic analysis [LK13; Eri08; Gru11; Hub08; Zap11; Zap12]. Gruzd et al. [Gru11]'s Twitter as an "imagined community" comes closest to our study, which found that Twitter is capable of forming a sense of community online for a single individual and their personal network. Our work provides a qualitative approach by interviewing many individuals participating in #TidyTuesday, providing insights into why data scientists participate in the project and how the social interactions lead to a successful CoP on Twitter which also provide facilitation of in-person events like meetups.

Recently, users of Twitter use daily hashtags which repeat on a given day and provide various means to form a community of practice. Daily hashtags have been used for sharing knowledge or organizing discussions for a particular topic or domain. For example, they have been used for academic advising with professional development [Pas19], and for synchronous discussions around healthcare [Gil16]. Within programming communities, #AdventOfCode [Was] is an example of a popular daily hashtag where each day of the Advent calendar presents a new programming puzzle to solve. Programmers then post tweets and share their thoughts about their solutions and reflections in solving the puzzles. The data science communities have also recently adopted daily hashtags using a similar structure for practice. #MakeoverMonday [Kri] is a daily hashtag for data scientists using the Tableau [Tab] software to produce data visualizations and post their submissions on Twitter every Monday. We extend the research on Twitter-based CoPs by studying #TidyTuesday as an example of a daily hashtag targeted towards data scientists wishing to practice their data science skills and become part of a community to grow in.

4.3 Method

In this section, we present details about #TidyTuesday and how it relates to similar projects in the R community and discuss the research questions we investigated, and the qualitative methods we used to answer them.

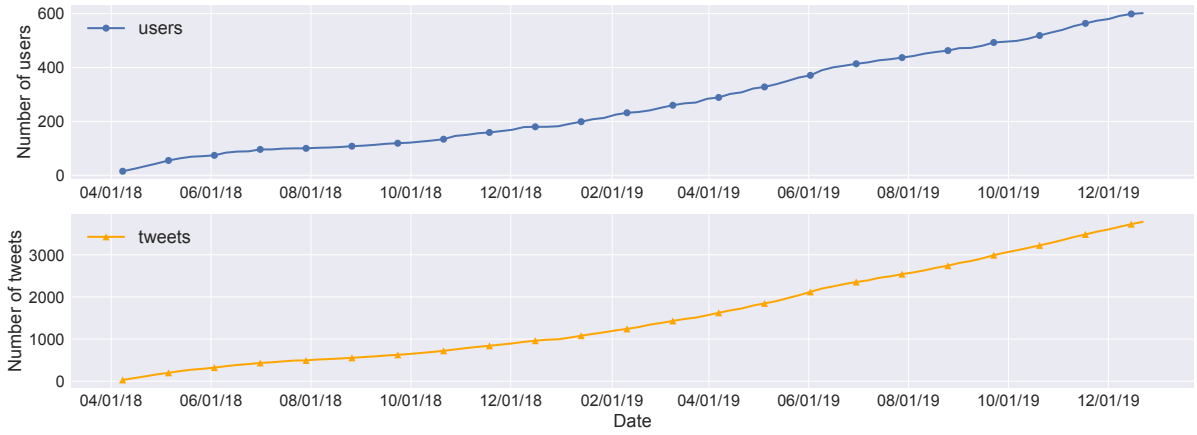

Figure 4.1 Cumulative growth of unique #TidyTuesday users and tweets from April, 2018 to Jan, 2020.

4.3.1 Research Setting: #TidyTuesday

4.3.1.1 Precursors to #TidyTuesday:

In order to understand the background behind #TidyTuesday, we first interviewed Thomas Mock, the creator of the project. During graduate school, Thomas became involved with an online learning community called R4DS (R for Data Science). Jesse Mostipak started the R4DS project in 2018 as a book club [Mos] for the R for Data Science textbook [Wic17]. Since then, the project has evolved into a learning space on Slack [Sla], where programmers of all skill levels can ask questions about R, similar to Q&A sites like Stack Overflow, but designed to foster a friendly, welcoming environment that promotes discussions. We gathered statistics from Jon Harmon who leads R4DS Slack and found that it has 6139 total members, 426 weekly active members, 130 of who have posted. There are also weekly office hours for learners who have more specific questions that mentors can answer.

When Thomas joined the R4DS online learning community, he wanted to start a smaller project designed to provide efficient practice for data scientists using R. The R4DS had experimented with a project called #TidyWeek [Tidb] with a different goal from R4DS: pairing learners and experts with an emphasis on reviewing code. Interested learners would sign up for #TidyWeek, who would be matched with a mentor to get help and feedback on their R project that they were working on. However, the matching process and coordinating between mentors and learners became too difficult to sustain and required a high level of coordination for both learners and mentors.

4.3.1.2 Inception and growth of #TidyTuesday:

Thomas then spearheaded #TidyTuesday, which was designed to be a smaller, loosely structured project that could solve the issues of #TidyWeek and his own pain points learning R. After privately experimenting with the project and getting feedback from the R4DS online learning community leaders (C13, I14, I17), Thomas introduced the project to others in the R community, who are

prevalent on Twitter (#rstats, #r4ds).[1] His main goal behind #TidyTuesday was to help himself and other data scientists practice their data wrangling and visualization by providing access to weekly released, real-world datasets, and to encourage sharing of code to facilitate social learning [Moc18]. The #TidyTuesday project has been steadily growing since its inception making it a suitable case study of how an online community of practice for data scientists can grow and sustain itself on Twitter. As shown in Figure 4.1, #TidyTuesday has become quite popular in the R community with over 3834 unique tweets that contained the hashtag from 607 unique users from April, 2018 (inception) to January, 2020 time period.

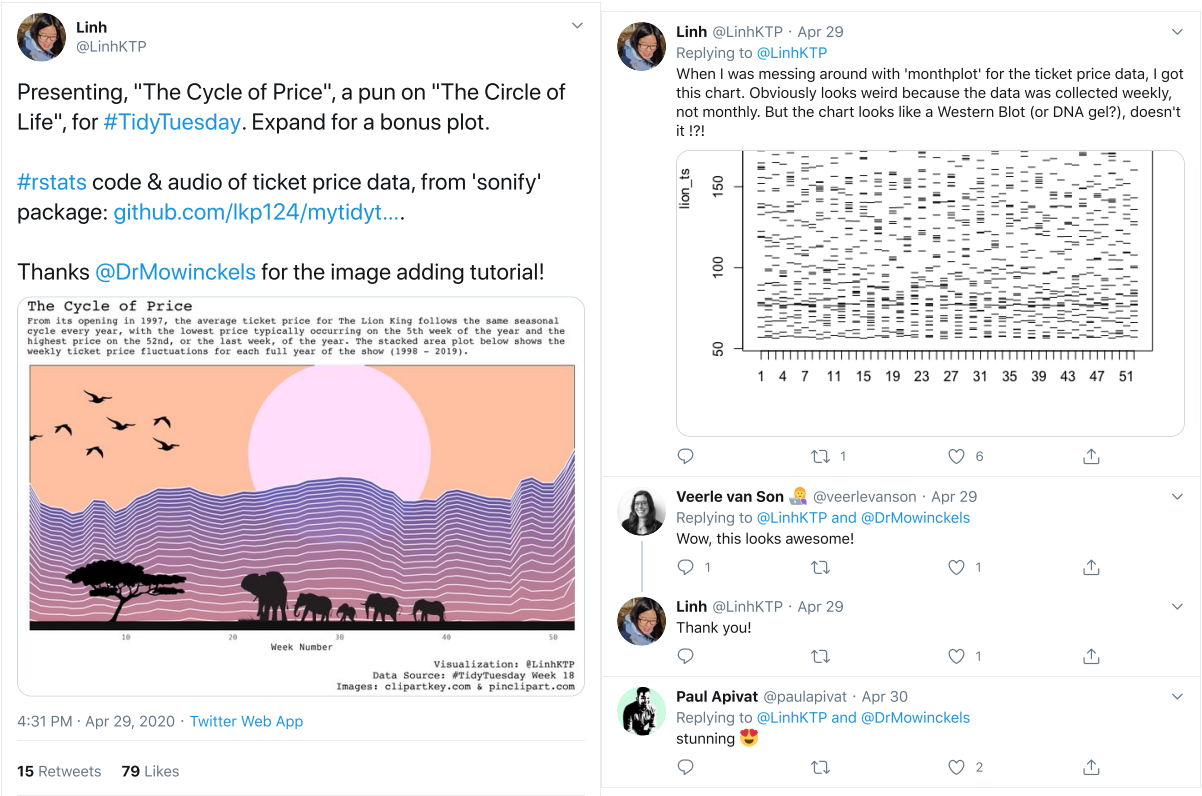

Figure 4.2 An example of a #TidyTuesday submission tweet and feedback.

4.3.1.3 #TidyTuesday tweet anatomy:

An example of a #TidyTuesday submission tweet and feedback are shown in Figure 4.2. On the left is the submission tweet where the poster provides background about the dataset they analyzed and visualized. To help others reproduce their work, a poster typically includes a link (GitHub) to the code and an attached visualization (stacked area plot) exploring a certain aspect about the dataset. Sometimes, the poster will provide a description of the R packages (`sonify`), functions

[1] `https://twitter.com/thomas_mock/status/980921600429252608`

or tricks which can facilitate disseminating best practices. Moreover, credit is given to individuals (@DrMowinckels_e) who have helped them produce the plot. On the right of Figure 4.2 is an example of what the feedback typically looks like, where others may compliment them and the poster may provide further elaboration on their submission.

4.3.2 Research Questions

In order to understand what the #TidyTuesday participants' motivations were, how the project benefited them, and how it formed an online community of practice, we had three main research goals. The first goal was to understand why a data scientist participated in the project. The second goal was to understand the beneficial experiences that data scientists went through as they participated in #TidyTuesday. Finally, the third goal was to identify the various ways in which #TidyTuesday helped form and sustain an online CoP on Twitter. Thus, we investigated three research questions:

- **Who participates in Tidy Tuesday and what are their motivations and goals?** To understand who typically participates in #TidyTuesday, we asked data scientists about their background, motivations and goals they hoped to accomplish with the project.
- **What do participants gain by participating in Tidy Tuesday?** To understand how #TidyTuesday benefits its participants, we asked questions about their overall experience with the project, including perceived benefits and challenges.
- **How does social activity around Tidy Tuesday cultivate a community of practice?** To investigate whether social activity around #TidyTuesday forms and sustains a CoP, we asked data scientists about their social interactions on Twitter and analyzed them through the lens of the CoP framework.

4.3.3 Interviews

Demographics and recruitment. We interviewed 26 total data scientists (Table 4.1). The participants were from 14 different fields and worked in 3 sectors: 13 in academia, 9 in industry, and 4 in healthcare. Participants self-reported age (18-24 = 1, 25-34 = 9, 35-44 = 9, 45-64 = 1, NA = 6), gender (14 men and 12 women), and education level (Bachelor's = 6, Masters = 6, Doctorate = 14). The participants had varying years of programming experience in R ranging from 1 to 3 (7), 3 to 5 (5), and 5 or more years (14). We recruited the participants using a combination of random and snowball sampling. We first used random sampling of the authors of tweets in the April 2018 to Jan 2020 time period and contacted them via email. We ensured that participants were actually making a submission for #TidyTuesday by inspecting their tweets; some were dropped and replaced due to unrelated tweets. For the posters group, we were interested in being able to contrast the experiences between different skill levels, so we tried recruiting between two groups: 7 *one-offs*, who only posted one submission and 8 *persistent*, who posted multiple submissions. To effectively recruit the curators

Table 4.1 Demographics of interviewees

Social Role *	ID	Gender	Degree	Field	Sector	Posts
Poster	P1	M	Masters	Data Science	Academia	1
Poster	P2	M	Bachelors	Statistics	Academia	6
Poster	P3	F	Doctorate	Environmental Science	Academia	1
Poster	P4	F	Doctorate	Library Science	Academia	1
Poster	P5	M	Doctorate	Data Science	Industry	1
Poster	P6	M	Bachelors	Data Science	Industry	27
Poster	P7	M	Bachelors	Biotechnology	Healthcare	17
Poster	P8	M	Bachelors	Biostatistics	Industry	10
Poster	P9	F	Doctorate	Math Education	Academia	6
Poster	P10	M	Masters	Marine Ecology	Academia	1
Poster	P11	F	Doctorate	Marine Ecology	Academia	11
Poster	P12	F	Doctorate	Ecology Science	Academia	10
Poster	P16	F	Doctorate	Statistics	Academia	15
Poster	P25	M	Doctorate	Radiology	Healthcare	1
Poster	P26	F	Masters	Statistics	Academia	1
Curator	C13	M	Bachelors	Content Science	Industry	50
Curator	C19	M	Masters	Marketing	Industry	15
Curator	C20	M	Doctorate	Data Science	Industry	24
Curator	C21	F	Doctorate	Data Science	Academia	14
Curator	C24	M	Bachelors	Bioengineering	Healthcare	24
Influencer	I14	F	Masters	Data Science	Industry	7
Influencer	I15	M	Doctorate	Data Science	Industry	43
Influencer	I17	M	Doctorate	Data Science	Academia	78
Influencer	I18	F	Doctorate	Data Science	Industry	22
Influencer	I22	M	Doctorate	Ecology Science	Academia	62
Influencer	I23	M	Masters	Psychiatry	Healthcare	72

[1] * Posters focus on posting and sharing their submissions with others. Curators make efforts to organize tweets and learning resources to make it easier for others to participate. Influencers grow and promote the movement.

and influencers, who were highly influential for growing #TidyTuesday and the R community at-large, we used snowball recruitment starting from Thomas Mock. We did not offer compensation for participants. To determine who to interview next, we used a constant comparison method [Gla67] to guide our decisions about theoretical saturation.

Social roles. From our preliminary examination of the #TidyTuesday tweets, and tying in past literature on the various skill-sets and roles played by participants within communities of practice, we categorized our participants based on three social roles: poster, curator, and influencer Table 4.1 presents the full list of participants and their social roles in the project. There were 15 *posters*, who varied in their level of engagement with #TidyTuesday from "one-offs" (P1, P3, P4, P5, P10, P25, P26) who only posted their submission once to those who posted subsequent posts (P2, P6, P7, P8, P9, P11, P12, P16). Liu et al. [Liu17] used a similar category for participants when analyzing

tweets related to the #ILookLikeAnEngineer identity hashtag movement; however, we do not focus on "passive" readers for this study since we are only interested in why members participated in the first place. There were 5 participants who served the role of *curators*: in addition to posting their submissions, these participants engaged in organizing, highlighting submissions or contributing new tools to enhance the project. Data curators have been studied before by Zagalsky et al. [Zag16] in Stack Overflow and mailing lists in the R community and by Middleton et al. [Mid20] for baseball analytics within the Sabermetrics community. In this study, curators were interested in improving #TidyTuesday and the larger #rstats community on Twitter by organizing tweets and creating tools to facilitate participation. Finally, 6 were *influencers* who were already immersed in the R community and had a large following on Twitter. They were a mix of individuals who were either highly involved in making contributions with their own #TidyTuesday tweets or spreading the movement and encouraging others to join. This social role is based on Graham & Wright [Gra14]'s study of "superposters" and the role they play in an online forum which found that, despite the potential for negative influence, they had a significant positive effect on others by helping them and being empathetic towards their problems.

Interview Protocol. The first author conducted 30-60 minute semi-structured interviews over Google Hangouts. All the interviews were done remotely using a template[2] which included questions around the following topics:

- Motivation to participate in #TidyTuesday.
- Experience participating in #TidyTuesday and its usage.
- Interactions with the R community on Twitter and elsewhere.

For each interview, the audio was recorded for transcription and analysis. The first author transcribed all of the interviews. We iteratively developed these questions based on a few pilot interviews to get meaningful responses around motivation, the experiences related to the daily hashtag, and community interaction. Topics like how participants engaged with the R community in contrast to other similar communities did not get much coverage; this is understandable as the R community might be their first one they engaged with. While the structure of the interviews remained constant, we let participants discuss other tangential topics.

4.3.4 Analysis

We audio-recorded and analyzed the transcripts of the semi-structured interviews. Before analysis, we first segmented the transcripts into different sections reflecting the semi-structured interview questions (Section 4.3.3). We then began analysis with open coding [Cha06] on each topic looking for similarities and differences across the interviewees' thoughts or actions and assigning short phrases as codes. Some examples of first-level codes include codes like "accountability", "copy-paste code", or "coding alone". We wrote memos and engaged in continual comparison of the codes with

[2] `https://go.ncsu.edu/tidytuesday`

one another and we performed focused coding [Cha06], grouping similar codes and analyzing them to identify high level themes like "creating rhythm for practice", "enhancing technical and communication skills", or "community participation on Twitter". Finally, we refined themes in the central concepts of participation in the project, the daily hashtag's impact and cultivation of an online community of practice.

4.4 Results

In our analysis, we found that participants had various intrinsic and extrinsic motivations, were positively transformed by participating in #TidyTuesday, and their social interactions on Twitter helped build a community of practice for data scientists using R. In the following sections, we discuss themes related to each research question.

4.4.1 Who participates in Tidy Tuesday and what are their motivations and goals?

Participants had various intrinsic and extrinsic motivates to participate in #TidyTuesday. Posters were mainly concerned with skill development and increasing their public presence in the R community through the project and their tweets. Curators, in addition to posting their submissions, were interested in organizing, highlighting submissions or contributing new tools to enhance the project. Finally, influencers were motivated to improve their skills, but wanted to focus their efforts on growing and promoting the project through various ways. We discuss the themes around motivation below and summarize them in Table 4.2.

4.4.1.1 Low barrier of entry:

To participate in #TidyTuesday, all participants were motivated by the low barrier to entry to contribute and get involved with the project. Participants expressed that it was easy to participate in #TidyTuesday because the requirements were minimal:

> *"To me, there's no better on-ramp then download R, download RStudio, install ggplot2 and then make your first plot. From there, when somebody makes their first plot they're like 'holy crap that is way better than me fussing around with Excel chart wizard."'* (C20)

The lack of a formal onboarding or signup process made it easy for participants to join the project when they felt ready. In fact, many participants (P5, P6, P7, P9, P12, I15) were unsure on participating and passively observed the project on Twitter for a few weeks (3-4), learning about how others tweet and interact with one another before they deciding to get involved. The *"lurking"* (P16) behavior corroborates previous work related to legitimate peripheral participation (LPP), where users are found learning about the community practices and behaviors given access to user profiles and shared artifacts [Hol19; Mar14; Bry05]. The choice of Twitter as the #TidyTuesday community public place provided similar affordances enabling LPP—participants can search, save,

Table 4.2 High level themes influencing participation

Theme	Representative Example
Low barrier of entry (Section 4.4.1.1)	"Instead of us having to come up with the idea of the data set Tidy Tuesday does that and it was just perfect." (P12)
Curated datasets for a time-boxed activity (Section 4.4.1.2)	"I had three hours a day where I was like on the subway essentially and it was the perfect thing to do during that time." (P16)
A weekly rhythm (Section 4.4.1.3)	"I kind of used it as a practice and a weekly sort of accountability." (P3)
Improve technical and communication skills (Section 4.4.1.4)	"Tidy Tuesday just seemed perfect because after a few weeks of just seeing a lot of people have done, you kind of pick up some tips about how to do stuff." (P6)
Connecting with the community (Section 4.4.1.5)	"I wanted to be a part of the R community first, and Tidy Tuesday with the visualizations was like a way to get there." (C20)

and observe others' tweets, which include links to the code on open source websites like GitHub, GitLab, or RPubs. Another aspect that attracted all participants to #TidyTuesday was that it is open for individuals of all skill levels, which helps achieve Wenger et al. [Wen02b]'s design principle to invite different levels of participation for growth of a CoP. Moreover, because the tweets are "easy access and public" (C20), it also satisfies Jones [Jon97]'s requirement that a virtual settlement should include a common-public-place where members can meet and interact.

However, there were participants who only contributed once or twice which suggests potential barrier of entry issues. P10, P25, P5, and P4 all had similar reasons as to why they haven't posted more submissions on Twitter. P5 and P10 did not post more submissions simply because they did not having enough time. P4 also had time constraints but they also explained that they spent a long time figuring out exactly what they wanted to do with datasets, and would've preferred some ideas to help her get started. P25 differs from the rest because they were not satisfied with the #TidyTuesday visualization that they worked on and therefore decided against sharing it.

Asynchronous submissions: We also found that participants were motivated by the fact that the project does not require any deadlines for a particular #TidyTuesday. All of the posters stated in one form or another that this asynchronous nature of the project offered flexibility. Because #TidyTuesday submissions don't have deadlines, several participants expressed that they felt less pressure finishing it on Tuesday (P4, P6, P11, P16). Sometimes the reluctance to participate on a certain week was because they *"couldn't find anything interesting about [the dataset]"* (I22). If they couldn't practice on a dataset a certain week, participants were assured by the fact that they could

always try the next #TidyTuesday. One participant even coined a new hashtag called #TardyTuesday to convey the fact that they don't usually complete their submission on Tuesdays:

> *"I think I coined Tardy Tuesday for a while because I never do them on Tuesdays. I get that if you do it on Tuesday, it's very defined and it's done. But Tuesday's a rough day of the week. I think on Thursday, I'd be more like able to sit down and do it but on Tuesday I'm either behind on something I need to have done or something else."* (P16)

4.4.1.2 Curated datasets for a time-limited activity:

Participants expressed that it was important that they had access to datasets every week that were manageable and could be timeboxed—allotting a fixed, maximum unit of time for an activity. As Sutton et al. [Sut18] have noted, large datasets can present multiple problems leading to "death by a thousand wranglings". Several participants (P6, P7, P16, I22, I17, I15) described that the datasets hit a *"sweet spot"* (I15) in terms of the size, making them attractive for a quick analysis. This aligns with all of the participants' treatment of #TidyTuesday as a side activity for extra practice. These manageable datasets allowed participants to limit their practice sessions yet be able to do exploratory analysis and visualization on interesting datasets *"and be done with it in two or three days"* (I22). I15 provided a nice explanation of these characteristics of the #TidyTuesday datasets that attracted them:

> *"There's a sweet spot. You need to have a dataset that answer a variety of questions. It might be between—I would say between—a thousand and a million rows is about the right size for Tidy Tuesday. In fact, it's usually less than a hundred thousand. You could do more but it's still the right area. You could download it quickly and analyze it. And the number of columns between 5 and 20. And if you jump into datasets in the wild, a lot of them won't look like that."* (I15)

4.4.1.3 A weekly rhythm to stay engaged:

Participants were also interested in the *"rhythm"* (I18) and/or consistency provided by new datasets released every week. P3 described the *"burst"* of activity on Tuesday and throughout the week, which is reflected in Figure 4.3 where there's a flurry of tweets on Tuesday, with lesser tweets the following days.

The #TidyTuesday activity provide "aliveness", a key component to cultivate CoPs identified by Wenger et al. [Wen02b] as an indication that the project is active to attract newcomers. Indeed, the *"consistent delivery of a dataset every week"* (I15) attracted participants with either the *"accountability"* (P3, P4, P11) factor or as a source of *"inspiration"* (P3, P16, I17, I22), to help them maintain their engagement with the project and form a practice regimen. Regardless of skill levels, the participants described they needed this routine to stay consistent with their practice.

In addition to rhythm, participants were also motivated by the anticipation of working with new and different types of datasets every week. For example, P3, P12, and C20 described feeling

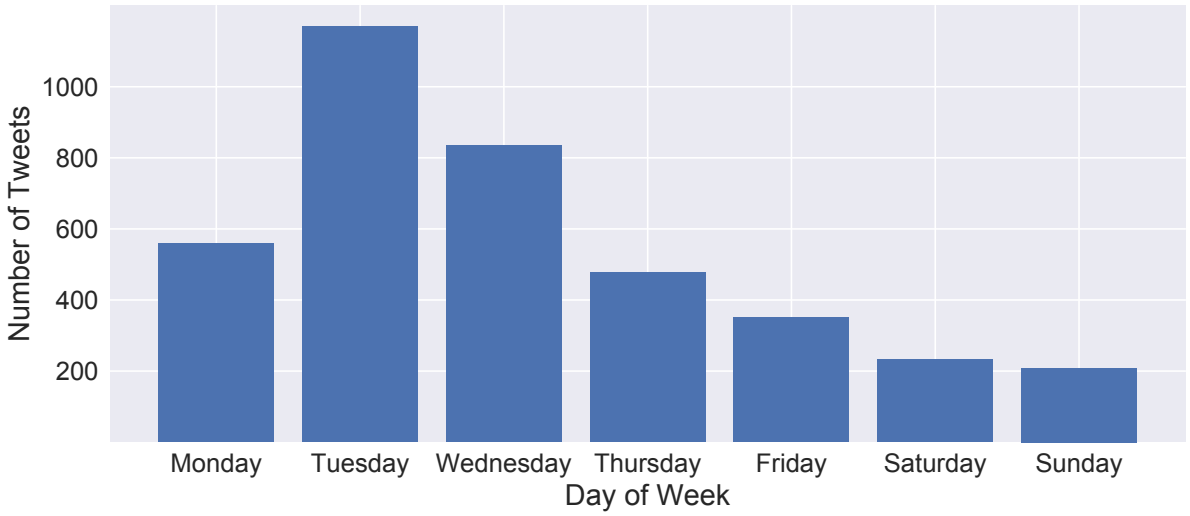

Figure 4.3 Number of #TidyTuesday tweets each day of the week from April, 2018 to Jan, 2020.

excited about the new dataset for each week, as well as the different types of analysis or visualization that were possible. P3 added that they were inspired by the intricate and diverse visualizations every week. Part of that is thanks to a core set of "super-posters" [Gra14] like I17 and I15, who consistently maintained high levels of posts (47 and 73 posts, respectively) with creative approaches on visualization datasets. Just like Cranshaw & Kittur [Cra11], we found this leadership helped participants (P8, C20, I22, I23) motivate themselves to begin making their own contributions.

4.4.1.4 Improve technical and communication skills:

All participants were interested in using #TidyTuesday to hone their technical and communication skills, or *repertoire* [Wen99], and build an online presence. In particular, learning was the most common intrinsic motivation between all participants, similar to developers in open source software by Ye & Kishida [Ye03]; participants in our study wanted to engage in "exploratory learning" and "learning by doing". For those who were just getting started with R (P2, P6, P7, P8, C20), traditional resources like classes, MOOCs, online tutorials or books were insufficient in providing adequate practice. They described #TidyTuesday as a push for improving their data science skills and improve their technical abilities to deal with real-world datasets instead of toy datasets built into R datasets.

Those who were already familiar with base R were interested in polishing their skills around the tidyverse style of R (P6, P9, P16, C20, I17, I22, I23) or pushing their data visualization skills. These participants either wanted to get familiarized with tidyverse packages or build on skills they don't normally get to exercise:

> *"I learned R before the tidyverse and then I was like, "Oh I should probably learn the tidyverse, I'm getting a little bit passé." So I went through all the main packages and was teaching myself the differences between base R and whatever like dplyr, purrr—all that*

stuff. My least favorite thing is actually making plots and ggplot is my nemesis so I was like okay this is perfect because it makes me practice making plots." (P16)

Participants (P2, P4, P16) were also interested in using #TidyTuesday as a forcing function for improving communication skills through blogging, making screencasts, or building a data visualization portfolio. This motivation has both intrinsic and extrinsic counterparts: participants wanted to improve their own personal skills while attracting the attention of others in the community and potential employers. They were also interested in data visualization techniques that they weren't normally used in their job. Influencers (I17, I22, I23) treated #TidyTuesday as a challenge to continually push the limits of data visualization skills using packages like `ggplot2`.[3]

4.4.1.5 Connecting and engaging with the R community:

Finally, all of the participants were motivated to participate because they wanted to become part of a community and engage in social learning. Similar to the motivation to improve their communication skills, participants had both intrinsic and extrinsic reasons to connect with the R community. Within McMillan & Chavis [McM86]'s sense of community (SoC) theory, they were seeking *membership* (feeling a sense of belonging) and *fulfillment of needs* (learning skills from others). For example, P2, P6, P16, and P9 described #TidyTuesday as a good motivator for both fitting into the R community and learning valuable skills from others:

"It helps learning a lot easier especially if you're trying to do it on your own. You're not really alone because you've got other people out there on Twitter or wherever sort of learning as well. I think the great thing about it is that it's this place where you sort of learn with other people even if you're not physically in contact with them." (P6)

The level of involvement participants wanted with the community was dependent on the participants' skill level and goals. Some of the posters and curators were newcomers to the R community (P8, P12, P16, P3, C20) who were motivated to join #TidyTuesday to absorb best practices and the norms of the R community, and increase their public presence online as part of their professional development. Put another way, #TidyTuesday became a way to audition for the R community on Twitter: *"I wanted to be a part of the R community first, and Tidy Tuesday with the visualizations was like a way to get there"* (C20).

The influencers who were already immersed in the R community were more interested in using #TidyTuesday as a way to *"give back to the community"* (I14, I15, I17, I22, I23) and helping newcomers have a welcoming experience. This fits under the SoC theory definition of *influence*, making a difference to a group. For example, I14, I17 and C13 were interested in fostering the welcoming culture and helping Thomas build the movement by participating themselves. For example, I17 accomplished this by participating heavily in #TidyTuesday to grow the movement and try to welcome beginners who are new to the project:

[3] `https://ggplot2.tidyverse.org`

"Self-motivated or self-directed learning is pretty easy for [some people]. But, I know a lot of people where they feel isolated, and it feels challenging for them. I just wanted to help. I saw the community and saw what was there and thought, "Hey this is neat, everyone's helpful and how can I help people learn R?" because if I help others learn R, I'll learn more R. I'll polish my skills and be a helpful guide in the community." (I17)

4.4.2 What do participants gain by participating in Tidy Tuesday?

#TidyTuesday transformed all of the participants regardless of their skill level both in regards to skill and professional development as well as getting more involved with the R community in general. We discuss these themes below and summarize them in Table 4.3.

Table 4.3 High level themes on the impact of #TidyTuesday

Theme	Representative Examples
Enhancement of skills through LPP (Section 4.4.2.1)	"I love looking at other people's visualizations and then reading their code and getting ideas on how I would build off of that." (P3)
Hashtags aided information retrieval (Section 4.4.2.2)	"I think I saw someone like have a package about you can make a gif or something and I was like wow that's a really cool package." (P4)
Expansion of skills outside of occupation (Section 4.4.2.3)	"Essentially, I just want to try some of the plots I don't get a chance to do in my research work." (P9)
Building an online presence for the job market (Section 4.4.2.4)	"The way I got that job was because I had all these blog posts and Tidy Tuesday stuff." (P16)

4.4.2.1 Enhancement of skills through legitimate peripheral participation:

All participants commented that reading others' #TidyTuesday code and visualizations helped them enhance their own technical and communication skills. Having access to others' work through their tweets and GitHub links facilitated legitimate peripheral participation for all participants, an important component of situated learning [Lav91] that helps explain how newcomers observed the project initially, then slowly participated by posting their own tweets. For example, P1, P3, P4, P11, and I22 practiced reverse-engineering skills by reusing and modifying others' code while working on their #TidyTuesday submission. Marlow & Dabbish [Mar14] noticed the same social learning mechanism in designers using Dribbble. Influencers like I22, I17, and I23 were pleasantly surprised by how posters *"borrowed and extended"* (I17) their code, giving credence to the effectiveness of

sharing code to transform peripheral members into experienced members. Since participants are primarily motivated by learning, this is in contrast to competition-fueled settings like Kaggle, where only a small minority of medium skill-level members shared and re-used code [Tau17].

Through #TidyTuesday tweets, participants also formed impressions about others' skills and their commitment to the project which helped them keep track of those with similar interests or skill-sets. This parallels Dabbish et al. [Dab12] and Marlow et al. [Mar13b]'s studies of GitHub, where they also found impression formation mechanisms using visible cues on GitHub like user profile and commits. We see similar visible cues with twitter profiles, tweets, and the visualizations. Among influencers like I17, I23, and I22, this enabled both learning *"creative approaches"* (I22) to visualizations and igniting *"friendly competition"* (I22). #TidyTuesday became a social learning tool for all skill levels to get inspired by others on what is possible with R and anticipate future posts from them:

> *"I love looking at other people's visualizations and then reading their code and getting ideas on how I would build off of that, but I haven't done a lot of them from scratch myself. I learn off of other people's awesome ideas that they share. Yeah that's a big part of it for me: I'm not looking to necessarily practice my skills as much as I am to be inspired and know what I can do based on what other people share."* (P3)

4.4.2.2 Hashtag as an information retrieval and R package discovery tool:

All participants mentioned that the #TidyTuesday hashtag allowed them to easily search for others' tweets and discover new packages in R. This need for searching others' submissions became such a common task, that a web application tool called Tidy Tuesday Rocks[4] was built to aggregate tweets and make them accessible in a central website. The hashtag made it possible to build the web app which helped participants (P4, P9, C20, I22, I23) search for past submissions that get buried due to high volumes of tweets or are difficult to find via Google searches:

> *"I've also used it to drill into the code, lift that off, and use it in my own. One of the maps I found on Tidy Tuesday rocks and I clicked [their] GitHub code and I was like, "Oh here's how [they] did it!" You see so many pages that are so heavily indexed over so long in Google that it's like really impossible to find what people are doing these days. I know that these things are recent because people did them over the last couple weeks."* (C20)

As C20 points out above, the #TidyTuesday hashtag became useful to find examples of R code that best reflects modern packages and techniques. In this way, the #TidyTuesday hashtag helped aid information retrieval and achieves Wenger et al. [Wen02b]'s "design for evolution" recommendation because it allowed aggregation of tweets based on the hashtag. Since the #TidyTuesday tweets were continually updated every week, participants were able to discover modern R packages that they had never used before:

[4] `tidytuesday.rocks`

> *"I find it a really good way to practice and try out new packages I've never used before. For example, for first time, I used the gganimate package for animating a plot, or ggalluvial package for doing flow diagram or alluvial plot."* (P6)

4.4.2.3 Expansion of skills outside of an occupation:

#TidyTuesday participants used the project to explore data visualizations that they don't normally get to make in their day-to-day job (P16, C20, I15, I17, I22, I23). Given the diversity of the datasets and the multiple ways of analyzing and visualizing them, #TidyTuesday became a *"choose your own adventure game"* (I17), which allowed participants to *"pursue something really weird"* (P16) beyond traditional visualizations:

> *"With my job, it's often kind of like longer-term projects. We're focused on this one specific thing. So in a given week, I'm not going to be working on mapping and NLP and animation and machine learning and stuff like that. So, it's kind of cool to have a different little thing to play with every week."* (C19)

R veterans like I22 and P9 commented how having access to the visualizations others produce also allowed them to challenge themselves and produce their own unique take on visualizations.

> *"I made a tree map just because it was really interesting. I think I got delayed and then came back like half an hour later and I was like "oh no some people have already used that idea!" But the only difference is you cannot interact with the tree map, and I thought I'm just going to add some additional things to it because it's gonna be boring to see two identical tree maps with different colors."* (P9)

Beyond R programming, the curators were able to use #TidyTuesday as a forcing function to improve other skills which was the same theme in Fiesler et al. [Fie17]'s study of an online fandom community. For example, C20 and C21 were able to use #TidyTuesday to showcase their skills making tools that help others find tweets or understand others' code. C13 learned how to start and maintain a podcast and its respective website, while C24 learned how to use screencasts for covering #TidyTuesday submissions.

Improvement of communication skills with both written and other media. Several participants also used #TidyTuesday to polish their communication skills through blog posts or screencasts, which as a side-effect gave others opportunities for social learning. Often times, participants would expand on their thought process behind the code and visualizations in the form of blog posts (P2, P4, P6, I22, I17, I15, I18). The blog posts helped improve the participants' communication skills, as well as reveal the decisions made behind the code or plot. Influencers like I15 and I18 decided to make screencasts to improve their communication skills and help others learn about how to read, wrangle, analyze and visualize datasets. This helped them improve their own communication skills, while providing additional learning resources for newcomers via LPP [Lav91]:

"The barrier to making a screencast is low. We already have quite an amount of written material. So using screencasts as an opportunity of getting another kind of medium out there and show how to use some of these open source packages was a good idea. Tidy Tuesday is a great example of another kind of content, and there are people out there who like watching videos and learn that way." (I18)

4.4.2.4 Building an online presence for the job market:

There were many participants who used their #TidyTuesday submissions to build a portfolio for data analysis and visualization (P2, P6, P9, P16, P26, C20, I15, I22, I23). As mentioned earlier, some participants wrote corresponding blog articles on their personal website. We found that tweets, blog posts, and visualization portfolios around #TidyTuesday provided the same transparency and acted as signals for employers as found by Marlow & Dabbish [Mar13a] with activity traces and visual cues on candidates' GitHub profiles. By posting their work online either on Twitter, GitHub, YouTube or their personal sites, several participants (P2, P9, P16, I15) were able to attract recruiters and were offered interviews or jobs:

"I wrote a post and I think a couple weeks later, there was a consulting firm on the East Coast. They're primarily doing a project in healthcare and they called me and said, "Hey we saw your interactive [plot] on Twitter and we would like to get to know you. We feel like you're gonna be a good fit for one of the senior consultant positions." I was like sure! I was really surprised that everything kind of happened because of that one tweet!" (P9)

This online presence even benefited experienced data scientist like I15, who made #TidyTuesday screencasts which employers used as convenient indicators of expertise:

"It's been awesome from an interviewing standpoint. People say how do you analyze your datasets. I'd say if you want to find examples, they are on YouTube. I did have one hiring manager who watched it all the way through and it definitely went well." (I15)

4.4.3 How does social activity around Tidy Tuesday cultivate a community of practice?

We found #TidyTuesday and the social interactions afforded by Twitter helped crowdsource knowledge, disseminate best practices in R, bootstrap offline events, and build an inclusive, welcoming community. We present the high level themes below and summarize them in Table 4.4.

4.4.3.1 Promoting modern R practices:

The #TidyTuesday project encouraged the use of the tidyverse packages, which were created to provide R with consistency and ease of use based on the idea of tidy data [Wic14]. Although all of the participants were familiar with this new style of R programming called tidy R, many had not made the transition yet. For example, P11, P6, P16 had significant experience in base R and made the transition to tidy R through #TidyTuesday:

"It's funny. Up until about a year ago, I honestly was on the base R side. But tidy R, it's so clean! I am fully committed now. I switched all my classes this last year to teaching tidy R in my undergrad classes too. There's just so much energy and effort that's being put into those packages and I think that's just like how it's moving. I still use [base R] but I moved almost completely away from for loops and things like that which I was completely attached to up until about a year ago. I would say really within the last year, I transitioned to being a more tidy coder." (P11)

Table 4.4 High level themes on community building

Theme	Representative Examples
Promote best practices (Section 4.4.3.1)	"Up until about a year ago, I honestly was on the base R side. But tidy R, it's so clean like I am fully committed now." (P11)
Curation to satisfy community needs (Section 4.4.3.2)	"I just kind of started annotating stuff for myself and then I realized, oh I bet other people would find this useful too." (C19)
Bootstrap offline events (Section 4.4.3.3)	"Bringing Tidy Tuesday from Twitter and grounding it in real life as hacky hours made sense." (P3)
Promoting an inclusive, welcoming community (Section 4.4.3.4)	"On Twitter I've kind of had to come out of my shell to post stuff but every time I posted things or interacted with people, they've just been so wonderful and supportive." (C19)

Through #TidyTuesday, participants adopted tidyverse packages like `dplyr` for data wrangling or `ggplot2` for visualization, which can be considered modern coding practices of R due to its rising popularity. Zhu et al. [Zhu16]'s study of Wikipedia found that adapting best practices are helpful when the target audience has had some experience already. Thomas Mock and the initial contributors like I17 were already versed in tidy R, and since most participants were at least familiar with the new style of R programming, it proved an effective promotion strategy. #TidyTuesday on Twitter supported "trendspotting" (continued development by keeping up) that Marlow & Dabbish [Mar14] discovered in their study on Dribbble.

4.4.3.2 Curation as a means to solve rising community needs:

The curators of #TidyTuesday have played a crucial role in enhancing the project and satisfying McMillan & Chavis [McM86]'s sense of community requirement of "integrating and fulfillment of needs" of the community. It also reflects the tendency of curation by the R community in other

channels like Stack Overflow and R-help mailing lists [Zag16], but adding new types of knowledge artifacts like web applications and interactive documents. In response to rising community needs, curators (C13, C19, C20, C21, C24) helped create packages and tools around # TidyTuesday related to organizing tweets, highlighting submissions, and walking through code.

Two packages were made to make dataset retrieval easier for any given week, and a web tool for browsing past submissions in useful ways. The `tidytuesdayr` package makes it easier to access #TidyTuesday datasets without leaving the RStudio IDE (Integrated Development Environment) [Rsta] by requiring a single line of code to load the right dataset from a particular week: `tt_data <- tt_load("2019-01-15")`. This solves the potential challenge for loading data, especially for beginners. Since it is difficult to search for tweets and remembers specific plots, a web application called Tidy Tuesday Rocks[5] allowed the community to browse past submissions by dataset or username. This tool also served as an *"R gallery collection"* (C20), which further inspired several of our participants (P3, P11, I17, I22).

To highlight past submissions and walking through others' code, curators created #TidyTuesday code walk-throughs, and a podcast. The package called `flipbookr` [Rey] was created as an interactive slide document designed to walk through `ggplot2` code, line-by-line on #TidyTuesday techniques. Similarly, a project called #TidyX [Hug] was recently created to provide screencasts reviewing past submissions and explaining the code step-by-step, as well as pointing out R packages and techniques for data wrangling and visualization. There are also annotations for #TidyTuesday screencasts which help viewers jump into specific timestamps for information about a particular R function or technique.

4.4.3.3 Bootstrapping offline events to provide engagement for learners:

Perhaps the most surprising use of #TidyTuesday was bootstrapping in-person events. Some participants (P3, P9, P11, P12) used #TidyTuesday datasets to facilitate in-person events. P9 used #TidyTuesday for organizing a hackathon meetup at their university, which they described as a *"pain-free process"* because #TidyTuesday gave them easy access to the curated datasets. P3 and P12 started a weekly social coding club called "hacky hours" for students at their university holding in-person meetups to alleviate students' fear of programming, and provide them with a welcoming learning space:

> *"I thought it was really important to have an activity or a place where students could just try different things and try learning functions in a social setting that is totally no stress. Or if they fail completely, that's fine and hoping that they would learn something and progress while having fun. So Tidy Tuesday seemed like the perfect thing to kind of center that goal and so we started our in-person Tidy Tuesday last spring. We've been sticking with it pretty much every Tuesday since during the academic year!"* (P12)

Similarly, P11 started a #TidyTuesday meetup with students at their university for similar benefits.

[5] `http://tidytuesday.rocks`

However, for P11, they had never worked on #TidyTuesday by themselves and started doing them with a group. P3, P11, and P12 all prioritized mentorship of students, encouraging experienced members to help the beginners in the group. This echoes the theme of enabling LPP for learners in Section 4.4.2, but in an offline setting:

> *"I spend the majority of the time helping others, less so working on my own plots because I wanted to facilitate learning. I'm trying to encourage some of the more senior graduate students now to take that role away from me a little bit more so that they can practice teaching others and starting to feel more comfortable teaching some of the newer graduate students and helping them decode issues."* (P11)

These experiences add a new perspective on Gruzd et al. [Gru11]'s study on Twitter as an imagined community and Wenger et al. [Wen02b]'s design components on cultivating CoPs. #TidyTuesday not only propped up an online CoP, but also facilitated in-person meetups, helping reduce efforts behind finding datasets, promoted learning via LPP, and allowed organizers to focus their efforts solely on logistics—setting up a calendar, planning the events, and choosing the locations. In effect, the in-person meetings combined the whole-community gathering taking place online (Twitter) which extends Gruzd et al. [Gru11]'s finding that Twitter can be an imagined community, but complemented an offline small-group gathering (meetups), which adds an additional layer of rhythm and meets [Wen02b]'s recommendation of public and private community spaces.

4.4.3.4 Promoting a welcoming, inclusive community:

The #TidyTuesday community welcomed people of all skill levels, coming from very diverse backgrounds, creating an online space to practice R together. All participants felt they felt welcomed into the larger R community through their participation in #TidyTuesday. Thomas played a large role of ensuring that newcomers felt welcome, and shaping positive behavior for #TidyTuesday through his moderation and example-setting:

Thomas expressed that he wasn't as focused on deterring certain behavior, but setting examples of positive behavior with his friendly replies on Twitter. This strategy has worked for #TidyTuesday and is backed by Seering et al. [See17]'s finding that users tend to imitate both pro- and anti-social behavior. Participants (P5, P8, P12, C20) described the immediate contrast when asking questions in other online channels Stack Overflow compared to asking on Twitter. The following sentiment resonates with Ford et al. [For16]'s study on barriers for participation on Stack Overflow which revealed that askers were hesitant in participating because they feared not receiving an answer back or receiving negative feedback:

> *"With Stack Overflow, everybody is looking to get points so you'll get people who will either not give great answers, but they want to get a point in, or you also get people who are very rude and very standoffish. If you ask how do you do something, instead of telling you an answer or how to get the answer, they'll kind of more or less insult you. We*

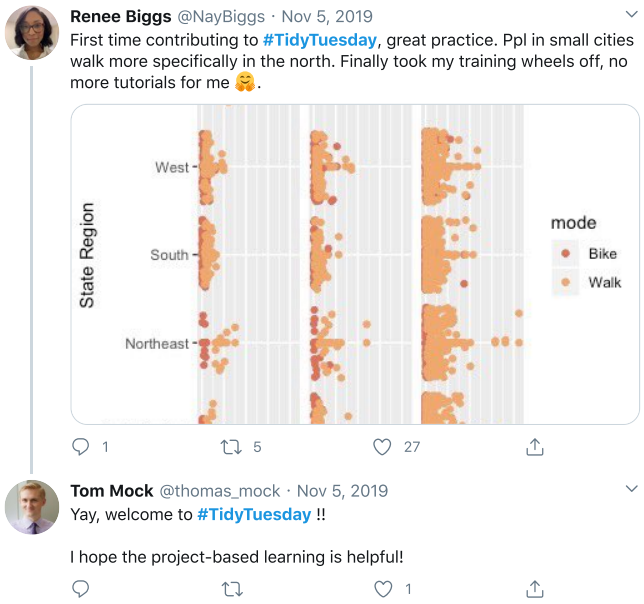

Figure 4.4 An example of Thomas welcoming a newcomer to #TidyTuesday.

> *know Twitter to be a very volatile place, but for whatever reason with the R community, it wasn't. People were always very helpful, always very nice and I just enjoyed it a lot more."* (P8)

Despite only receiving passive feedback of likes or retweets, several participants (P6, P16, C19, C20) were surprised by the amount of positive feedback from the community. Within the SoC theory [McM86], by participating and receiving feedback, participants felt *membership* (feeling like they belong) and *influence* (mattering to its members). For example, P16 was encouraged to continue posting more #TidyTuesday tweets because they weren't used to receiving such feedback in their academic setting:

> *"I think mutual support of just like every time I get a like, I'm like, "oh somebody thinks that I did something cool!" I think that's the big thing in grad school that you also don't get—positive feedback frequently. So I know it's dumb to smile when I get a like on my tweet but I do."* (P16)

Sometimes, there was an *"happy accident"* (Thomas) of interactions between participants and highly renowned data scientists in the R community. For example, P11 recounted how Hadley Wickham liked one of their students' #TidyTuesday post on Twitter, which was very encouraging for the student.

#TidyTuesday also helped several participants increase their online presence. Through their contributions in #TidyTuesday, P3, P11, P12, and C19 dramatically increased their online presence and made an impact by helping others get involved with #TidyTuesday. P3, P11, and P12 commented that #TidyTuesday helped them mentor students and encouraged them to become more active

online and reduce their fear of sharing work online, especially for those who were introverted. For C19, they described their transformative experience when they shared their annotations of #TidyTuesday screencasts with the R community on Twitter:

> *"I'm a pretty introverted person so even on Twitter I've kind of had to come out of my shell to post stuff, but I mean every time I posted a #TidyTuesday tweet or interacted with people, they've just been so wonderful and supportive. I've never seen any place on Twitter where people say, "This is so helpful! Thank you! Great job!" It's so amazing that the people are like this on Twitter. So it's been just a wholly positive experience."* (C19)

4.5 Discussion

In this section, we discuss benefits and challenges behind #TidyTuesday and provide both the R community and similar communities with guidelines on how to effectively use a daily hashtag to build an online community of practice. The guidelines fall under three broad categories: barriers to entry for beginners, technological improvements to facilitate better learning, and social interactions to form and sustain a welcoming, inclusive online community.

4.5.1 Lowering the barriers to entry

In the following subsections, we discuss some barriers to entry for #TidyTuesday and provide suggestions on how to provide better onboarding.

Can I participate? There were some data scientists who didn't realize #TidyTuesday was designed for everyone or weren't clear on the skill requirements to participate, even though everyone was welcome. For example, one programmer with minor experience in R asks:

> *"People who participate in #TidyTuesday: how much experience with R/coding in general did you have before doing it the first time? I've tried to do this week's task but I'm finding myself pretty lost."*[6]

This tweet suggests a barrier to entry issue related to getting started in #TidyTuesday, and the implicit skill requirements for a beginner. Steinmacher et al. [Ste15] identified several relevant social barriers that stops students from participating in open source software (OSS), which included barriers like "newcomers need orientation" and "technical hurdles". Potential #TidyTuesday members might have difficulty knowing what skills they need to participate, which has been noted as a problem in Cranshaw & Kittur [Cra11]'s study of Polymath Project, an online collaboration between mathematicians solving open problems. P4 suggested including beginner prompts for datasets about potential actions or questions the analyst can explore which addresses the barrier of "finding a task to start with" [Ste15]. As for "technical hurdles", beginners in #TidyTuesday might face the

[6] `https://twitter.com/scottjdavies01/status/1201751167782408192`

challenge of the tooling required for #TidyTuesday, such as Git/GitHub for code sharing. To help lower this potential barrier to entry, P12 suggested a small tutorial for learning the bare minimum required for Git/GitHub. Learning resources are linked at the bottom of the Tidy Tuesday GitHub Readme file, but it might make the resources more visible if placed at the beginning, so that a newcomer can better orient themselves to the project.

Reduce fear aversion for novices. Beginners wishing to participate might also suffer from a case of *fear aversion* after witnessing intricate code and stunning visualizations produced by experts. These submissions could either inspire or hurt learners. To reduce this fear, a possible remedy is for experts to point out they are also learners who sometimes struggle to produce visualizations for #TidyTuesday. As discussed in Section 4.4.2.1, some of the posters included a blog article or a screencast corresponding to their submission to provide further explanation behind the code and the plots. We believe these *"learning out loud"* (I14) activities done by experts can help beginners by highlighting the gradual, incremental steps taken towards producing those complex, creative plots:

> *"Those [blog posts] are awesome because people think that [experts] easily come up with these polished, awesome work. What the plots doesn't show is that, 'No, we failed and we did like 1500 experiments to get to where we are' or 'there's like a pile of sketches on paper on my desk."'* (I17)

We observed this initiative to blog and record screencasts only among a few of our participants (P4, P16, C24, I22, I15, I17), who were specifically interested in enhancing their communication skills (Section 4.4.1). We agree with C20 in encouraging experts in the community to learn out loud to help newcomers feel more comfortable participating in online social coding projects like #TidyTuesday.

4.5.2 Better mechanisms for practice and learning

Provide diverse forms of resources. #TidyTuesday was effective in sharing knowledge and building an online CoP primarily because participants championed the idea of code-reuse and explanations through tweeting, blogging and making screencasts. Participants provided external, in-depth explanations in the form of blogs (P4, P6, P16, I17, I22) or screencasts (I15, I18). As we discussed in Section 4.4.2, only the influencers made #TidyTuesday screencasts, and it's important to note that they are experts in R. C20 and I22 suggested encouraging shorter screencasts to nudge the less experienced members of the community to take part creating video formats to showcase tips and tricks in R to further promote social learning:

> *"Stepping through somebody's code doesn't quite get you there. You don't get to hear their design choices or like why they split it up and into these two different things. I would love for the people who are creating these super stellar visualizations to take 10-15 minutes and go back and describe how they got there."* (C20)

Faas et al. [Faa18] have found that live stream coding can support the growth of learning-focused communities that mentor both the streamer and each other during and after streams. Influencers

such as I15 and I18 have provided screencasts which can help learners pick up "tricks in R" (I15). Live streams could further enhance the learning experience by allowing questions in-situ, a feature which blog posts and pre-recorded screencasts lack.

Twitter as a platform to promote friendlier learner interactions. The Twitter platform provided a friendly experience for #TidyTuesday to learn and practice R, compared to other channels like Stack Overflow or R-help mailing lists. For example, beyond questions and answers, Twitter replies can initiate discussion threads which are discouraged on Stack Overflow, while still allowing "participatory knowledge creation" [Zag16]. Unlike the aggressive behavior found in R-help mailing lists, #TidyTuesday participants found very little aggression on Twitter, a surprising finding that we will further discuss in Section 4.5.3. All participants expressed that the R community on Twitter (via #TidyTuesday and #rstats) has a welcoming attitude towards beginners, allowing follow up discussions and simple questions without any fear of scorn or negative comments. I14 pointed out that *"this was not always the case"* and the leaders of the R community such as R-Ladies Global and RStudio have helped changed the culture. However, I14 also expressed caution that R-specific community forums—such as RStudio Community[7]—could potentially lead to the "posting is hard" barrier to entry on Stack Overflow [For16] by requiring questions to be structured a specific way (for e.g. using a `reprex`[8] for a reproducible example). We suggest the R community and other online communities to consider SNS sites like Twitter to form a community of practice that allows casual dialog, ongoing discussion threads, and friendly interactions.

Limitations of Twitter as a platform for online learning. While #TidyTuesday participants received positive feedback, they did not always provide constructive feedback. Kou & Gray [Kou17] studied *distributed critique*, a set of critique practices whereby geographically distributed creators engage in the critique of design artifacts and processes. We sometimes see evidence of this type of interaction (Figure 4.5). However, constructive feedback was rare among our participants. There has been an attempt to resolve this with a related hashtag called #RFeedbackFriday, to explicitly ask for feedback, but P6 commented, *"I tried it but did not receive any feedback and it may have fizzled out."* Another online CoP for healthcare has encountered similar difficulties on Twitter [Gil16]. Twitter is a public space and restricts tweets to 280 character limit, so it may stifle meaningful feedback required by learners because of a fear of publicly criticizing others, or not allowing depth. We echo Wenger et al. [Wen02b]'s suggestion for a "backchannel" for private conversations might help to provide deeper interaction between members of the community, using applications like Slack.

Better infrastructure to facilitate feedback and mentorship. To better facilitate deeper interactions between learners and mentors, we believe a better technology infrastructure is required that explicitly focuses on receiving feedback and mentorship. C21 engaged in the #MakeOverMonday—a daily hashtag for visualizations using Tableau—and expressed how the project has a central site that is used for critique by experts in a webinar. Such efforts require time, energy and funding which is beyond the scope of a voluntary effort like #TidyTuesday. However, we suggest taking advantage

[7] `https://community.rstudio.com`
[8] `https://reprex.tidyverse.org`

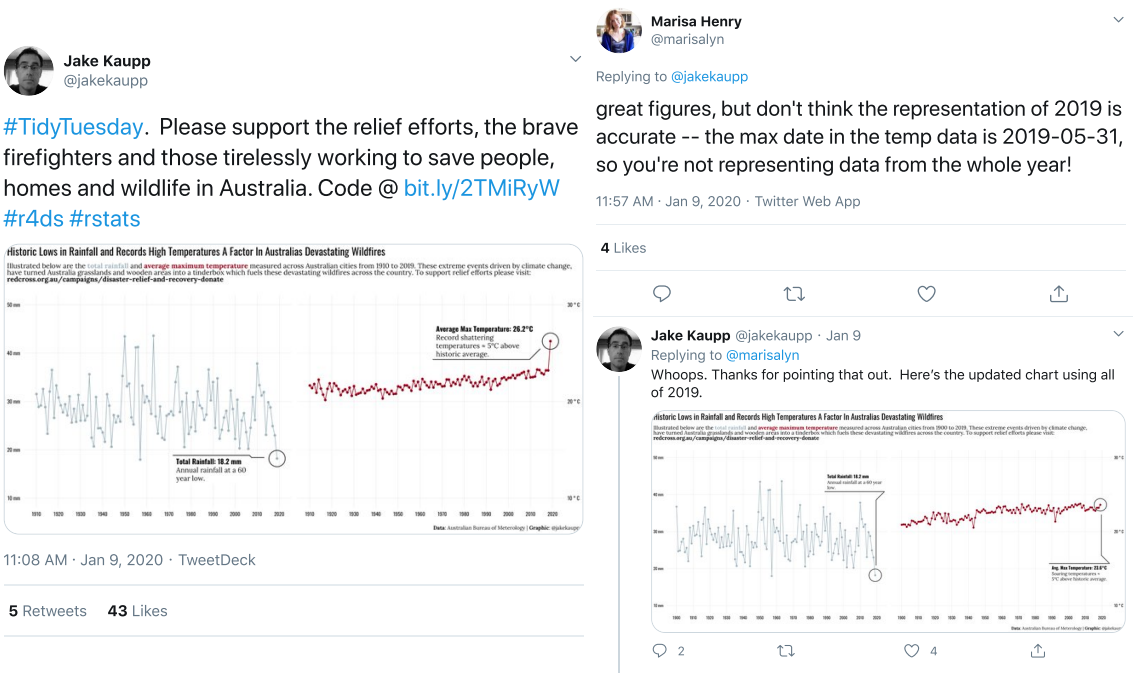

Figure 4.5 An example of constructive criticism on a #TidyTuesday submission tweet.

of a central place like the #R4DS online learning community on Slack, which could help support the one-on-one conversations, or Ford et al. [For18]'s just-in-time mentorship learners to inform cultural norms. Alternately, one could draw inspiration from Xu et al. [Xu14]'s system Voyant, which allows users to get feedback of their designs from the crowd. A similar system could be re-purposed for data scientists so that they can receive feedback on their plots or code.

4.5.3 Organically growing an online learning CoP

Choose technologies that support an open structure for growth. We believe a big part of the success behind #TidyTuesday was Thomas' decision in keeping the core structure of the project loose—a weekly dataset and a few rules to participate—yet allowing others to build on the project by making it open source. In other words, Thomas "designed for evolution" which is recommended by Wenger et al. [Wen02b] to organically grow a CoP. #TidyTuesday accomplished this well by choosing the right technologies: Twitter and GitHub.

Hosting #TidyTuesday on GitHub helped Thomas maintain the project for free and make use of crowdsourcing efforts for finding new datasets, fixing problems regarding those datasets, or making improvements to the project. Through GitHub's issues, Thomas was able to get help on fixing uploaded datasets[9], or project-related resources like information in the documentation[10]. However, P16 pointed out that a potential issue for a single moderator is burnout, a state of exhaustion caused by excessive and prolonged stress. This is similar to the burnout identified by Fiesler et al.

[9] `https://GitHub.com/rfordatascience/tidytuesday/issues/186`
[10] `https://GitHub.com/rfordatascience/tidytuesday/issues/162`

[Fie17] for experienced coders, which can be an issue for a project maintainer. To combat this, P16 suggested adopting the idea of "RoCur" or Rotating Curator: rotating the spokesperson on a social media account, where every week, a different member of the community manages the Twitter account, sharing their their views on using R, as well as tips and tricks. This is directly inspired by the @WeAreRLadies[11] effort out of the R-Ladies Global. For #TidyTuesday, a RoCur candidate might be a highly motivated individual such as a super-poster [Gra14], a curator or an influencer, to help curate/clean a dataset and interact with and promote others' tweets.

As mentioned in Section 4.4.2, using Twitter as the sharing platform for #TidyTuesday submissions enabled curations of various forms. Firstly, the hashtag tagging mechanism helped grow the movement by supporting information retrieval (Section 4.4.2.2), accruing knowledge and forming links to various learning resources. Hashtags became such a useful mechanism for growing #TidyTuesday that it sprung two new daily hashtags from our participants (#TardyTuesday and #Tidydors). Having access to submissions via Twitter also allowed curators to organize the tweets and/or artifacts produced by posters as well as provide further pedagogy on particular submissions for learners. As the project evolved, members started becoming aware of particular challenges of #TidyTuesday and sought to improve the project using their skills outside of R programming alone. Since Thomas welcomed anyone to help fulfill these needs, some members in the community jumped on the opportunity. For example, in Section 4.4.2, we mentioned the benefits of having access to others' code but passively reading code might not be helpful for learners. Curators stepped in to solve this issue by either highlighting specific packages and techniques (C13), walking through the code line-by-line (C24), or showing the incremental evolution of the code (C21).

To maintain and sustain the growth of an online community, we encourage keeping the core structure loose, and choose technologies that can help promote contributions and promote curations to enhance the project.

Encourage experts and influencers to engage with newcomers. The success behind #TidyTuesday in fostering an inclusive, welcoming online community of practice on Twitter owes a large part to the involvement of Thomas, influencers and other leaders within the existing R community. At the rstdio::conf 2020 [Rstc], Kate Hertweck delivered a a talk about how R communities are unparalleled in their inclusivity and commitment to learning collectively [Her20]. She noted several solutions to reduce barriers of entry like managing expectations and creating interest through expectation of activity and continuity. We believe Thomas has accomplished expectations by simply promising a dataset every week, leaving the task of analysis and visualization open-ended. Thomas also used the #TidyTuesday hashtag to create rhythm (Wenger et al. [Wen02a]) within the community and helps create what Kate recommended as "FOMO", the fear of missing out.

We also want to highlight the importance of the larger R community and the key players mentioned in Section 4.2.3 which provided a solid foundation for promoting inclusivity and diversity of members for #TidyTuesday. With the existing #rstats and #R4DS communities on Twitter, which

[11] `https://twitter.com/WeAreRLadies`

embraces this culture promoted by stakeholders like RStudio, R-Ladies Global, rOpenSci[12], Thomas was able to carry this spirit forward with the #TidyTuesday project by example-setting positive behavior [See17]. In a recent tweet, a user asked:

> *"I'm curious as to how the R community came to be so supportive and welcoming (as opposed to so much of the tech world). Anyone have ideas? #rstats"*[13]

A number of responses followed including Hadley Wickham and Jenny Bryan who provided an insight as to why Twitter is becoming a welcoming and inclusive space for the R community. Hadley commented *"that this wasn't at all the case 10 years ago"* and that perhaps *"each shift to a new space (r-help -> SO -> twitter) can be accompanied a refocusing of shared goals"* or *"it's just dominated by founder effects and shifts tend to be led by younger folks who are more in touch with initial pain of learning."* Indeed, in Section 4.4.3.4, we mentioned how Hadley liked a #TidyTuesday post by P11's student. Receiving the attention of leaders leads to what McMillan & Chavis [McM86] calls *membership* (sense of belonging) and *influence* (mattering to the group) within the sense of community theory. Jenny added that the *"growing role of twitter and @RLadiesGlobal creates space for new voices (vs "sorry all spots were filled 10 yrs ago")."* This type of leadership and initiative seems unique to the R community and is embraced within the new space on Twitter, a contrast to other channels like Stack Overflow, which has presented several barriers to entry, especially for women [For16].

However, to prevent *"insular"* (P16) data science communities, as suggested by I15 and P16, #TidyTuesday could be opened up for the Python community[14], where users are adopting the tidy data framework [Aug16; Hou; Yan]. Hence, #TidyTuesday can benefit other similar communities and help them cultivate their own online community of practice around #TidyTuesday, uniting even more data scientists in their efforts to become experts.

To organically evolve a new online CoP over time, we find that influencers and leaders of the community can play a vital role in growing the community by engaging and interacting with members of all skill levels, and instilling the feeling of being part of a community.

4.6 Limitations

Our analysis of the #TidyTuesday project represents an initial understanding of the dynamics and nature of participation in daily hashtag, social coding projects. We studied #TidyTuesday for the R community using a qualitative approach through semi-structured interviews. However, there are some limitations to our approach that represent opportunities for future research.

Generalizability. Our findings are drawn from one daily hashtag out of several others targeted at data scientists. For example, we did not study #MakeOverMonday, which is another daily hashtag

[12] `https://ropensci.org`
[13] `https://twitter.com/OwenChurches/status/1254634256472485896`
[14] `https://www.scipy.org`

serving a different need: using the Tableau software to create data visualizations instead programming using R. Hence, the themes we derived around motivations, skill development, professionalization, and community growth only represent the participants' experience at this moment in time for #TidyTuesday. It is important to note that some aspects of our findings might be unique to this project and the R community. Future research should explore other data science communities to improve our understanding of how to use daily hashtags as a tool for growing an online community of practice.

Participation bias. Another potential limitation of our study is a self-selection bias in our interviewee sample. Our sample was selected to contrast the experiences of people who participated in #TidyTuesday but, we do not know the experiences of *readers* [Liu17; Ant10] who passively engage by viewing, liking, or retweeting tweets. As a result, we may miss out on important issues related to barriers of entry. Participants also knew the study was about a discussion of the #TidyTuesday project, which may have influenced our sample towards those with the strongest feelings about the project. We mitigated this issue by using random sampling for all of the posters and curators and recruiting people of different skill levels and engagement with #TidyTuesday. However, we do not know the feelings or experiences of non-respondents and can only compare their tweets and participation levels.

Qualitative method. The challenges and recommendations we provide for daily hashtags as a way to provide an online learning CoP are based on participants' perceptions and experiences of #TidyTuesday, not quantitative indicators of the hashtag's effect on their behavior. To derive themes, we used qualitative coding to analyze and interpret our data which is limited by theoretical sensitivity and the synthesis conducted by the researchers participating in that process. We followed the guidelines set by Carlson [Car10] and performed a single-event member check with our results. 22 participants replied and agreed with our results and requested minor changes to their quotations or demographic information. Future studies should examine quantitative aspects of a daily hashtag project such as the dynamics of the tweets, or how the project spread on Twitter. Rosenberg et al. [Ros20], for example, have started an investigation of the posters' code itself, which offers insights on code evolution over time as an indicator for skill development. These quantitative measures can be useful to characterize #TidyTuesday, but we believe our themes provide rich insights and offer new directions for future work to further understand the benefits and challenges associated with the use of daily hashtags.

4.7 Conclusion

In this study, we conducted a qualitative case study on #TidyTuesday—a social coding project for data scientists using R—using the framework of CoP, and extending previous work related to forming and sustaining online CoPs on Twitter. From our analysis of semi-structured interviews with 26 participants, we examined motivations and goals of data scientists participating in #TidyTuesday and how it benefited them. We found that the participants were attracted to the rhythm provided

by the project, the opportunity for professional development, and becoming part of the larger R community. Through #TidyTuesday, participants enhanced both technical and communication skills by learning from others, adopting best practices in R, and building an online presence. #TidyTuesday was effective in forming an online CoP by disseminating best practices, providing opportunities for curations to satisfy community needs, bootstrapping offline events and promoting an inclusive, welcoming community. Based on our findings, we discussed several benefits and limitations to using daily hashtags on Twitter to form a community and provide guidelines on cultivating a successful CoP using a daily hashtag such as placing a low barrier of entry for newcomers by providing onboarding, normalizing the sharing of code and artifacts to promote social learning, and making room for evolution for organic growth and sustenance. We believe daily hashtags can be adopted by other data science communities interested in cultivating an online CoP.

CHAPTER

5

INTERACTIVE EXPLORATION OF DATA SCIENCE CODE

In the previous chapter, we studied how an online community of practice helps programmers learn data wrangling by browsing and extending others' code. How could we extend support within Integrated Development Environments (IDEs) for data science programming in a way that helps support this behavior of foraging and adapting others' code? In this chapter, we discuss a tool we built called Unravel, an in-situ exploration and learning tool within the RStudio IDE for data scientists using R. We discuss the results of a user study [Shr21a] evaluating the usefulness of the affordances of Unravel for understanding, exploring, and debugging data wrangling code.

5.1 Motivation

Data scientists have adopted a popular design pattern in programming called the fluent interface for composing data wrangling code. The fluent interface works by combining multiple transformations on a data table—or dataframes—with a single chain of expressions, which produces an output. Although fluent code promotes legibility, the intermediate dataframes are lost, forcing data scientists to *unravel* the chain through tedious code edits and re-execution. Existing tools for data scientists do not allow easy exploration or support understanding of fluent code. To address this gap, we designed a tool called Unravel that enables structural edits via drag-and-drop and toggle switch interactions to help data scientists explore and understand fluent code. Data scientists can apply simple structural edits via drag-and-drop and toggle switch interactions to reorder and (un)comment lines. To help data scientists understand fluent code, Unravel provides function summaries and

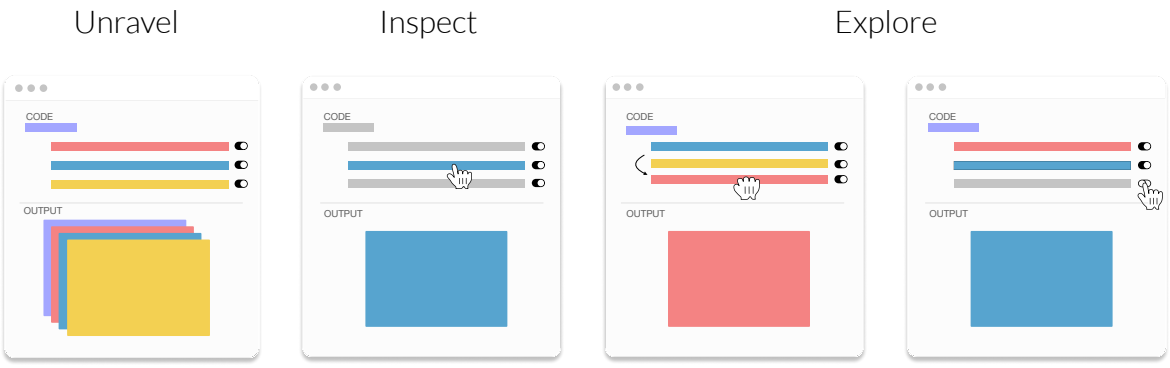

Figure 5.1 Unravel is a tool that helps data scientists understand and explore fluent code via structured edits using drag-and-drop and toggle switch interactions. The data scientist unravels fluent code to get access to intermediate outputs for each line. They can then inspect a particular line of code and its respective output. Data scientists can explore the code using drag-and-drop to reorder lines, and toggle switches to enable or disable lines and automatically produce new outputs to investigate.

always-on visualizations highlighting important changes to a dataframe. We discuss the design motivations behind Unravel and how it helps understand and explore fluent code. In a first-use study with 14 data scientists, we found that Unravel facilitated diverse activities such as validating assumptions about the code or data, exploring alternatives, and revealing function behavior.

5.2 Introduction

Data scientists apply a common programming design pattern—the fluent interface [Fow05; Fow10]—when they transform and wrangle data tables, or *dataframes*. The distinguishing feature of the fluent interface is that it composes multiple operations into a chain, with each operator in the chain accepting data from the result of the previous operator, performing a computation on it, and passing its result on to the next operator. Advocates for the fluent interface suggest this style of programming improves readability by removing the need to assign intermediate results to variables. To illustrate how fluent interfaces are applied in practice, consider fluent code written in R (Figure 5.2).

In the "tidyverse" [Tida] dialect of R, the pipe (%>%) operator forms the links in the chain of expressions by piping [Pip] results of function calls together, shown in Figure 5.2. In Figure 5.2a, the `penguins` variable containing the dataframe from the Palmer Penguins dataset [Hor20] gets piped (%>%) or passed to a `select` function to select columns, `species` and `flipper_length_mm`. The result is piped to the function `group_by` function to group the data by the `species` column which is finally piped to the `summarise` function to calculate the mean of the flipper length (`flipper_length_mm`) according to each species and store it in a new column, `mfl`. Note how the final code is built up by solving smaller subproblems (selecting, grouping, and summarising), forming a single large chain. But, what if there is a problem with the final output?

Although fluent code is designed to be readable and concise, these advantages come with a cost:

```
penguins %>%
  select(species, flipper_length_mm) %>%
  group_by(species) %>%
  summarise(mfl = mean(flipper_length_mm))
#> Output:
#>   species    mfl
#> 1 Adelie      NA
#> 2 Chinstrap  196.
#> 3 Gentoo      NA
```

(a) Summarizing mean flipper lengths of penguin species.

```
penguins %>%
  select(species, flipper_length_mm) %>%
  # group_by(species) %>%
# summarise(mfl = mean(flipper_length_mm))
#> Output:
#>   species flipper_length_mm
#>   ...                   ...
#>   ...                   ...
#> 4 Adelie                 NA
```

(b) Inspecting the line up to `select` with a dangling `%>%`.

Figure 5.2 An example of exploring fluent code in R, which outputs a dataframe of mean flipper lengths of different penguin species.

isolating problems within a broken chain becomes a clerical and cumbersome process. Because fluent code removes intermediate variables, one of its significant disadvantages arises when the data scientist needs to inspect an intermediate result. In Figure 5.2a, a data scientist might be surprised to find the mean of flipper lengths of the Adelie and Gentoo species are `NA`s or "not available." To hunt for clues, the data scientist is forced to "unravel" the chain and find the "broken" link (Figure 5.2b), where they have to modify and re-execute the code to discover `flipper_length_mm` contains missing values, an easy to miss detail. To verify the source of `NA`s, the data scientist had to comment lines, remove a dangling pipe operator, and execute the code up to the `select` function. They can then fix this issue by excluding rows with `NA`s for `flipper_length_mm` could be removed by inserting a function called `drop_na` before the `select` line.

We identified several limitations behind existing solutions to help data scientists explore fluent code. Prior research has focused on helping data scientists become more productive by managing messy code [Hea19], keeping track of versions [Ker17b], exploring alternatives [Wei21] or generating code [Dro20] using programming-by-example techniques. However, these tools are designed to help manage entire scripts or notebooks, and do not provide affordances to easily understand and explore code at a finer-grain level. In the R community, data scientists have voiced a need to support introspection and debugging tools for fluent code, with several attempts at solutions.[1] Existing inspection and debugging tools attempt to solve parts of the problem but fall short in several ways. Current solutions require a data scientist to meticulously debug, log, and selectively execute of code. For example, a debugger [Bac14] is a heavyweight solution for exploring fluent code and it forces the data scientist to linearly step through their code. Printing intermediate results to the console—for example, with `tidylog` [Elb21]—is lightweight but generates noisy output. Other solutions require newer types of operators to debug fluent code which might introduce more issues.[2] This suggests a need for easily exploring and understanding fluent code in data science.

To address this need, we introduce Unravel, a tool that enables structured edits using drag-and-

[1] `https://community.rstudio.com/t/whats-currently-the-recommended-way-to-debug-pipe-chains/14724`

[2] `https://win-vector.com/2017/01/29/using-the-bizarro-pipe-to-debug-magrittr-pipelines-in-r/`

drop and toggle switch interactions with always-on visualizations to help data scientists explore and understand fluent code. Unravel is a web application that runs within the RStudio IDE. Using an interactive code overlay, data scientists can *unravel* a chain of fluent code in R to examine intermediate dataframes, understand the transformations of dataframes along the chain, and apply simple structural edits without typing. Data scientists can apply structural edits to fluent code by reordering lines using drag-and-drop, or enabling or disabling lines using toggle switches. To help data scientists understand each step in fluent code, function summaries describe the transformations on dataframes and always-on visualizations are used to highlight important changes to dataframes. We designed Unravel to help data scientists get clarity on data transformations, and reduce the burden of typing to manipulate fluent code. The contributions of this paper are:

1. A tool called Unravel that enables structured edits via drag-and-drop and toggle switch interactions with always-on visualizations to help data scientists explore and understand fluent code. We discuss the design motivations behind Unravel and how data scientists can use it to support a variety of tasks.

2. Through a first-use study with 14 data scientists, we demonstrate that Unravel complements an IDE workflow, offers an interactive way to explore fluent code, and supports a variety of tasks related to understanding and writing data wrangling code.

Asha begins her investigation on checking why the final dataframe was grouped in the final output (Figure 5.3 C). Unravel focuses on the final line with the `mutate` function automatically, and Asha notices that the `percent_male` and `ratio` on the code and the dataframe output are marked as visible changes, but the `year` column is marked as an internal change. She had assumed`summarise` would have removed all group variables after calculating the sum of `year` and `sex` columns, but a group variable was kept. Puzzled, Asha investigates the summary of the `summarise` line to figure out how it handles group variables (Figure 5.4).

The new `total` column is marked as a visible change, while the `year` has stayed an internal change. To her surprise, Asha learns from the summary that `summarise` will drop the last grouping variable (`sex`), but keeps the rest of the group variables (`year`). This explains why she only saw the `year` column marked as internal change in the final output. Before moving on, Asha wants to confirm that `summarise` works on a grouped dataframe, so she temporarily disables the `group_by` line (Figure 5.5). The `summarise` line is automatically focused. Asha glances at the dimensions the dataframe output which is only one row and column. She confirms that without `group_by`, `summarise` will work on the entire dataframe instead of particular columns.

Asha now wants to sample the first `year` and `sex` groups. She adds a line in the editor to select the the first group using a `slice` function, placing it before the `group_by` line. Upon running Unravel on her new code, she comes across an error (Figure 5.6). Examining the dataframe dimensions on the `slice` and subsequent functions, she realizes she had only selected the first row of the original dataframe. Asha fixes the issue by reordering the `slice` line below the `group_by`, which

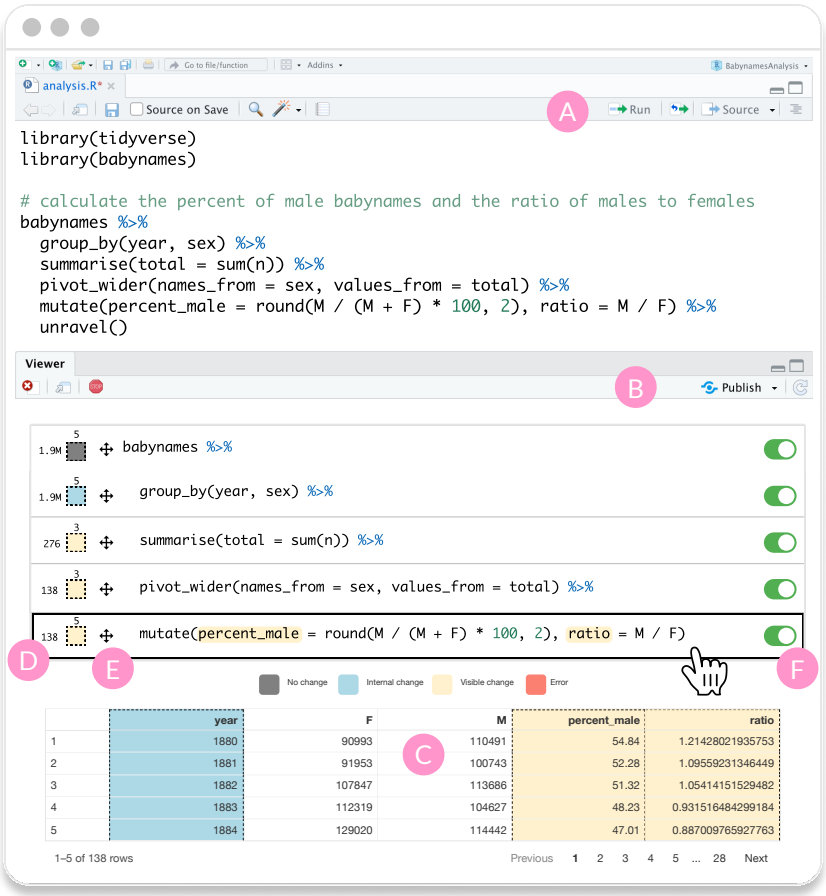

Figure 5.3 The workflow and interface of Unravel.

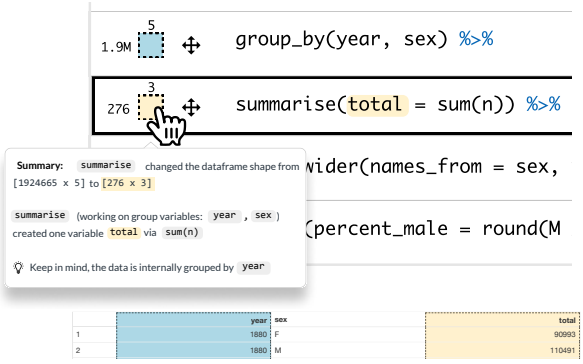

Figure 5.4 Visual highlights on the code, function summary, and output are applied when focusing on a line.

automatically updates the output to return the first `year` and `sex` group (Figure 5.7). She also learns that `slice` will keep the groups `year` and `sex`.

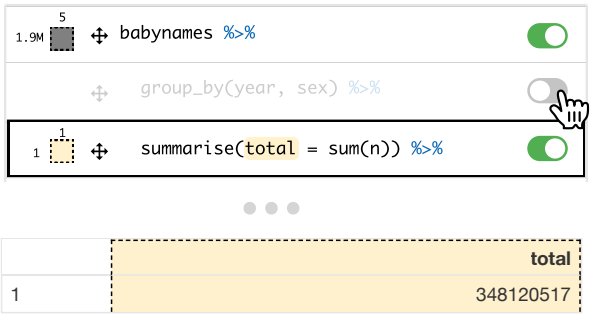

Figure 5.5 A line can be disabled using the toggle switch which automatically re-evaluates the remaining lines.

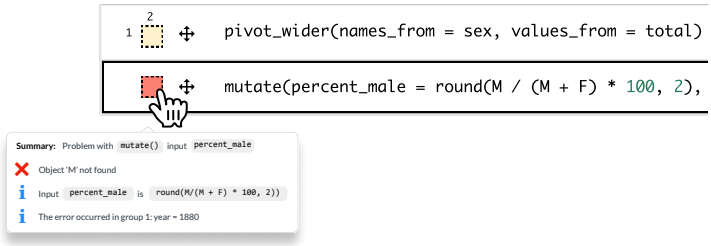

Figure 5.6 Clicking the summary box of the line with an error displays the error message.

5.3 Related Work

Unravel builds on prior tools that help data scientists write and understand code. Our work is closely related to the research on interactive tools that enable exploratory programming.

Writing code for data science. In computational notebooks, researchers have developed tools that help data scientists write and modify code. For example, Gather [Hea19] helps analysts find, clean, recover, and compare versions of code in cluttered, inconsistent notebooks. While Gather was designed for the notebook as a whole, our tool helps data scientists manage messes that arise within fine-grained, fluent code chains. To explore alternative code in notebooks, Fork It [Wei21] introduces a technique to fork a notebook and directly navigate through decision points in a single notebook. We designed a more lightweight approach to explore code by allowing exploration through overlays and structural edits on the code itself, for example, by enabling, disabling or reordering lines. To help data scientists generate data wrangling code, Wrex [Dro20] uses programming-by-example. Similarly, mage [Ker20] is a tool that helps users generate code based on the modifications made from interacting with dataframes. Unravel complements tools like Wrex and mage by allowing data scientists to understand and iterate on the machine-synthesized code.

Understanding code for data science. Prior work has explored tools to help data scientists understand code. For example, Wrangler [Kan11] is an interactive tool designed to ease the process of writing data transformation scripts. Wrangler takes a table-centric approach where data scientists

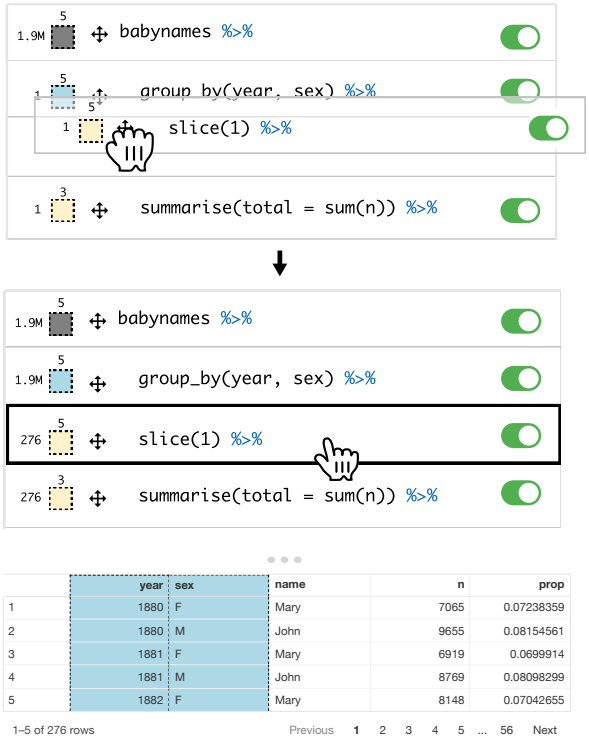

Figure 5.7 Drag-and-drop can be used to reorder a line, which automatically re-evaluates code to generate new outputs.

manipulate the table to produce scripts; our approach assumes that data scientists are already working with code and supports their understanding and exploration through it. Lau et al. [Lau21]'s TweakIt is a system designed to help end-user programmers collect, understand, and tweak Python code within a spreadsheet environment. Unravel shares similar design goals such as previewing outputs on different parts of the code, but it surfaces these capabilities through interactive visual overlays on the fluent code chains.

Closely related to our current work is Pu et al. [Pu21]'s system for animating dataframe wrangling and visualization pipelines in R. Datamations automatically animate fluent code in R using `tidyverse/dplyr` [Tida] packages and provides a paired explanation and visualization of each step in the chain. Our work differs in multiple respects. First, Datamations provides visualizations of operations within the chain with summaries on the intention behind each step. In contrast, Unravel provides an interactive tool that allows direct access to the intermediates throughout the chain for further inspection and exploration. While Datamations provide basic visual cues for tabular animations—such as highlighting a column for different grouping variables—we provide visual cues for more information such as the intermediate dataframe shape, and the type of change occurred at each step. Lastly, Unravel allows for exploration within the code, dynamically creating explorable code upon structural edits, whereas Datamations provides animated explanations.

Interactive affordances for code exploration. Researchers have investigated many useful inter-

active affordances for code explorations that we adapt to the data science context. Although these affordances were designed for different contexts, they are useful for addressing some of the pain points regarding fluent code.

Always-on visualizations can help data scientists understand and inspect code and data. Lieber et al. [Lie14] built an IDE tool called Theseus which provides an always-on visualization to display the number of API calls made within the editor for JavaScript code. We adapt this visualization technique for fluent code to display dataframe properties like its row and column dimensions. Similarly, "projection boxes" [Ler20; Fer20] are an always-on visualization technique for displaying runtime values of Python programs such as the contents of arrays. This can help minimize context-switches when writing data wrangling code since it requires a constant checking code and dataframes. Unlike projection boxes, we display one intermediate dataframe at a time instead of displaying them all at once.

Interactive debuggers and steppers are another useful technique for exploring data wrangling code. For example, Whyline [Ko04a] introduced an *interrogative debugging* interface for asking "why" or "why not" questions about a program. Whyline visualizes answers in terms of runtime events connected to the questions. Although our tool does not directly support asking explicit questions, it can aid this type of investigation by facilitating inspection of each intermediate dataframe in fluent code for data wrangling. Timelapse [Bur13] is a tool that helps web developers browse, visualize, and explore recorded program executions using debugging tools such as breakpoints and logging. We record program executions on fluent code to support investigation of all intermediate dataframes produced in the chain.

5.4 System Design and Implementation

Unravel is a tool that is run within the RStudio IDE to support data scientists introspect and explore fluent code using R. We picked R because it is widely used in data science, and is a popular language in data science based on the TIOBE index.[3] [Cha21] and HTML/CSS/JavaScript.

5.4.1 Design Motivations

Fluent expressions are used by many programming languages in data science. For example, LINQ (Language-Integrated Query) is a fluent interface in C#, a convenient wrapper—known as an object-relational mapper (ORM)—around database query languages like SQL. Languages like Python use the fluent interface for data analysis code through the `pandas` library. In the R community fluent code encompasses a vast majority of existing R code. The fluent pattern is used in a collection of R packages called the "tidyverse" [Tida] to facilitate importing data, wrangling data, computing statistics, manipulating strings, and modeling data. In the RStudio Community Forums, a popular

[3] `https://www.tiobe.com/tiobe-index/r/` The R language also provides metaprogramming capabilities which make it convenient for some stages of the implementation such as parsing and evaluation of intermediate expressions. It is built using the R Shiny Framework

Q&A site for R, the tidyverse is the 3rd largest category suggesting users experience pain points with these packages on a daily basis. Among the various contexts where fluent code is used, we examined data wrangling as an important activity to support because it is one of the most time consuming and difficult aspects of analysis [Mul19; Das03].

We examined the R community to identify pain points expressed by data scientists when understanding and exploring fluent code. Data scientists have expressed the need for transparency about the data that they are transforming.[4] One data scientist expressed how "we aren't good at tracking state"[5], and it's easy to miss whether or not a dataframe is grouped, where "working on a grouped [data] that you forgot is grouped can lead to 'unexpected' results"[6]. To inspect issues in fluent code, a traditional debugger can be too heavyweight for exploring smaller code snippets. The data scientist also has to linearly progress through their code and cannot openly explore code at any step. The R community has explored special pipe operators to debug fluent code, but these can introduce more typing mistakes and confusion for data scientists by adding more syntax to remember.[7] `tidylog` [Elb21] is a lightweight solution which prints the summaries of functions to the console output, but this can generate noise and it does not save intermediate dataframes for further inspection. During explorations of the code, data scientists have to constantly switch between the source editor and the console output to validate the effect of code on dataframes. This forces context switches. Altogether, we identified a need for an in-situ tool within an IDE—such as RStudio—that provides clarity on transformations, and reduces the burden of typing to manipulate fluent code. To address these needs, we arrived at the following design goals:

D1. Provide transparency about the dataframe in fluent code. The code and the respective dataframe intermediate outputs should be accessible at all times. Users must be able to click the relevant part of the chain to view its intermediate dataframe and glean basic information like row and column dimensions, the types of changes occurred and a summary about the transformation.

D2. Allow just-in-time explorations of fluent code. To help data scientists easily perform inspections on fluent code, they must be able to perform simple structural edits to the fluent code. Structural edits must instantly update the UI to easily explore the new intermediate dataframes.

D3. Minimize context-switching to unravel fluent code. To minimize context-switching, the tool should be integrated into data scientists' workflow within the IDE. Users must be able to input their own code to explore the chain.

[4] `https://community.rstudio.com/t/whats-currently-the-recommended-way-to-debug-pipe-chains/14724`

[5] `https://twitter.com/mjskay/status/1367244873607249922`

[6] `https://twitter.com/aosmith16/status/1369689345335070732`

[7] `https://win-vector.com/2017/01/29/using-the-bizarro-pipe-to-debug-magrittr-pipelines-in-r/`

5.4.2 Implementation

We present the implementation of Unravel by describing the entire process from invoking the tool to exploring a code snippet in the web application. We discuss our design decisions for all of the features to support our design goals in Section 6.2.1.

5.4.2.1 Code Parsing and Trace Executions

Unravel initially parses the user's code (Figure 5.3 A) and splits the fluent code into multiple code snippets that represent each part of the chain. Unravel parses the code passed to the `unravel()` function. We make sure to check the abstract syntax tree (AST) to ensure that the code is fluent code using the %>% operator and that it contains at least one line of code, a variable (or symbol in R) pointing to the dataframe. Unravel then splits the fluent code into intermediate expressions of the chain on the %>% operator. For each expression, we strip %>% operator at the end of an expression in order to evaluate it We store a list of these expressions to be evaluated in the next step. A challenge we faced was graceful handling of parsing errors along the chain. We chose to perform a best effort at parsing syntactically correct lines until it hits a problematic line, excluding the rest of the lines. This simpler implementation relies on the user to fix their code first instead of skipping to the lines after the error.

Trace executions. To produce the intermediate dataframes, we iterate through the list of intermediate expressions from the previous step, and evaluate each expression to create a new list holding all of the intermediate dataframes. For lines that throw an error, we store the error message and skip the rest of the lines that may follow. The message is used presented in the function summary tooltip to give users feedback as they would receive it on the console output. An alternate implementation could be to skip the line which causes the runtime error, and keep evaluating the rest of the lines. We decided to rely on a simpler solution: storing the error message displaying it when the user clicks on the summary box (Figure 5.3 D) next to the problematic line so that they can try to fix it. The user could also toggle switches to disable problematic lines. Unravel extracts and stores its row and column dimension information, as well as the type of change occurred. As before, these dataframes and the associated information are stored in a list for the UI to reference.

Summary generation. Unravel generates summaries using an extension of `tidylog` [Elb21], a package that is designed to log function summaries of tidyverse R code onto the console output. By loading our extension of `tidylog`, we overriding transformation functions with custom logging functions of using the same name and signature. When the user calls functions like `group_by`, we use the custom functions to introspect into the input dataframe and its arguments. We extended `tidylog` to capture summaries of each function instead of printing them to the console. There are numerous ways one could describe a function's effect on a dataframe such as warnings against certain parameters. However, we decided to focus on three simple pieces of information: 1) Mention the dataframe dimensions and if they have changed, 2) Highlight important column variables, and 3) Provide supplementary information about functions that have subtle changes like `summarise`

(Figure 5.4).

5.4.2.2 Visual Cues

Before interaction is possible, Unravel constructs the GUI using information about the chain from previous steps. To provide transparency about the data and its lineage in fluent code (D1), Unravel uses the dataframe dimensions, types of changes, and function summaries to create visual cues. We first describe the design of the visual affordances below.

Data change schema. To help data scientists pay attention to subtle changes, we designed a simple data change schema which visualizes different types of changes. We analyze the difference between the incoming and the resulting dataframe when a transformation function is called. Different colors are used highlight changes within the summary box, code and the dataframe output. "No changes" are means no change occurred after an operation. "Visible Changes" means the dataframe was transformed (e.g. creating new columns, mutating existing columns). The "Internal Changes" means there is no visible effect but the dataframe has been marked as a grouped by variables. Finally, the "Error" is to indicate a runtime error.

Code, summary, and dataframe callouts. To help data scientists keep track of dataframes and their state, Unravel highlights the code, function summaries, and the dataframes using the data change schema described above. We drew inspiration of this design by Wayne [Way]'s strategy to use run-time information to highlight parts of the code. There are lots of properties one could access from a dataframe during runtime, such as the number of missing values, but we decided to highlight column variables of interest in the code, output, and function summary (Figure 5.4). For a particular line, Unravel compares the previous and the new dataframe to highlight column names if they were transformed, or used as a group variable. Unravel also highlights text related to the changed column variables within the function summary text. Finally, the output dataframe column(s) is also highlighted accordingly.

Always-on visual cues for data transparency. To achieve our design goal of providing transparency (D1) about the dataframe, Unravel uses always-on visualizations. Data scientists have to continually track properties about dataframes which can be cognitively demanding, especially in complex data wrangling code composed of many operations. TensorSensor [Par] approaches the problem by improving the quality of exception messages around data dimensions, a particularly difficult task for novices. We were also inspired by Lieber et al. [Lie14]'s always-on visualizations tracked the number of api calls for web applications to help prevent misconceptions among students. Unravel's always-on visualizations consists of a summary box next to each line which displays the dataframe's row and column dimensions (Figure 5.3 D) and highlights to indicate the type of change occurred. Visual diffs—a display of the differences between lines of code—could have been used to illustrate differences between two dataframes. However, data scientists might not always be interested in checking the change between operations and an always-on diff visualization might be too disorienting. We decided on a simpler design to only show the snapshot state of intermediate dataframes.

5.4.2.3 Fluent Code Interactions

Unravel constructs the GUI by incorporating the dataframe information captured by evaluating intermediate expressions from the previous steps. We also link communication between R and JavaScript to respond to user interactions. Once the setup is complete, users can start inspecting the fluent code or apply structural or code edits for exploration. Below we discuss the design behind the structural drag-and-drop and toggle switch interactions to achieve D2.

Fluent code overlay. To help data scientists interact with fluent code, Unravel creates a web application within RStudio which overlays the code with a UI (Figure 5.3 B). Lerner [Ler20] used the idea of live projection boxes—presenting runtime values in boxes as the user types—for live programming to keep track of changes in data types like lists and arrays. We considered adopting this idea for data wrangling with fluent code, but typing can be cumbersome and the continual visual updates could become distracting. We designed Unravel as an *exploration mode* for data scientists to inspect and explore fluent code in isolation from their other code. Therefore, Unravel is presented in a separate window but within the IDE. Unlike projection boxes, Unravel only shows one dataframe at a time for a particular line in the fluent code. Information about the intermediate lines of code and their respective dataframes are used to populate and update UI elements on the fluent code overlay for displaying dataframe dimensions and types of changes occurred.

Structural edits via drag-and-drop and toggle switch interactions. To help data scientists easily edit fluent code, Unravel provides drag-and-drop to reorder lines and toggle switches to enable/disable them. The order of operations (lines) is important in fluent code because a dataframe is transformed by functions in sequence along the chain. Drag-and-drop interactions on code has been used previously to help users refactor or change code, and fix bugs [Lee13; Bar16] We use drag-and-drop to explore the effects of function order. Using the move icon (Figure 5.3 E), a line can be dragged before or after another line. Upon dropping a line, Unravel automatically evaluates the code to produce new dataframes to explore. Unravel will also handle trailing `%>%` operators for the last enabled line in the new code overlay. Another structural edit we implemented was enabling or disabling a line using toggle switches (Figure 5.3 F). We use toggle switches as another structural edit to help data scientists examine the effects behind the presence or absence of certain functions. Although a simple edit, this can be useful for isolating the exploration on certain lines of the chain.

5.4.3 System Scope and Limitations

We limited the scope of our tool in order to explore the usefulness of interactive exploration of fluent code. Here we briefly describe the scope and limitations of Unravel.

Supported code. We scoped our tool to focus on single-table data wrangling functions in the `dplyr` [Wic21] and `tidyr` [Wic19b], data wrangling packages that use the fluent interface. Unravel is limited to fluent code. Certain non-fluent code like variables storing dataframes and other similar data types like lists could benefit from always-on visualizations. Another issue is that some functions can have parameters that also accept a function as its value. For highlights on code, Unravel is

limited to simple function parameter values representing column names, not parameter values which are themselves functions. Finally, the output of an operation in fluent code could produce a list. However, Unravel only supports dataframes as the output of code and would need to be extended to render different types of data.

Evaluation limitations. Unravel does not attempt to sanitize a valid fluent code for side-effect functions, guarantee deterministic outputs, or optimize for performance. Some functions in both base R and `tidyverse` R cause side effects instead of returning values. Unravel is not currently aware of such functions, which could cause unexpected results. Using our evaluation strategy to generate the intermediate outputs, if a line within the chain contains a function that generates random numbers (e.g. `runif`), we currently generate new numbers for each subsequent operation. This can be an unexpected result if programmers make use of such functions. Lastly, we did not optimize Unravel to handle large dataframes and opted to use smaller datasets for the study. Hence, Unravel will become sluggish once dataframes become too large.

5.5 Evaluation: First-Use Study

To evaluate the usefulness of Unravel, we conducted a first-use study with 14 data scientists, 6 from academia, and 8 from industry. Participants had varied levels of experience. On a 5-point Likert scale, participants self-reported their experience in data science ($\mu = 3.6$), data wrangling ($\mu = 3.7$), R ($\mu = 3.7$), and the fluent interface ($\mu = 4$).

5.5.1 Methods

We conducted the studies over video conference using an online version of Unravel. We began each study by describing the tasks to participants. The tasks used built-in R datasets like `mtcars` and `iris`, some open datasets like `diamonds` (included in the `ggplot2` package [Wic16]), `babynames` [Wic19a], and `gapminder` [Bry], as well as one hand-crafted dataset called `student_grades`. The participants were tasked with exploring several code snippets written in tidyverse R dialect using the `dplyr` and `tidyr` packages. The code snippets were chosen to tease out how users would discover and explore the chain associated with prototypical types of data wrangling pipelines like selecting, filtering, mutating, grouping, and summarising dataframes.

For each code snippet, the task began open-ended where participants could explore each code snippet with Unravel then focused on specific tasks tailored to each snippet. We wanted participants to start using Unravel with open-ended exploration to capture their initial interactions with the tool. We then focused on specific tasks related to probing certain lines, and performing actions like toggling lines on or off, reordering lines, and asking them to observe the effect of the functions on the dataframe. Finally, we also asked participants to explore their own code in the RStudio IDE to gauge how well Unravel could be integrated into their daily workflows. While interacting with the tool, we asked participants to think aloud and ask questions. After the completion of the study, we administered an exit survey to measure the usefulness of Unravel features, and to ask for any

additional comments from participants about their experiences.

5.5.2 Post-study Survey Results

On 5-point Likert scale, participants positively rated the usefulness of Unravel overall ($\mu = 4.6$). Participants found the clickable lines for viewing intermediate dataframes ($\mu = 4.6$) and toggle switches for enabling or disabling lines ($\mu = 4.6$) were the most useful features. Similarly, the participants positively rated the usefulness of the summary boxes for viewing the dimensions and data change type ($\mu = 4.4$), and drag and drop for reordering lines ($\mu = 4.3$). However, there were less positive ratings for the usefulness of the data change color schema for visual callouts on code and dataframe outputs ($\mu = 4.1$), and function summary tooltips ($\mu = 3.9$). 93% of the participants responded that they would likely use Unravel to debug fluent code, while 79% of the participants responded that they would use it to understand fluent code.

5.5.3 Qualitative Results

We present our qualitative results from the user study, describing the interactions we observed, and the feedback participants provided throughout the tasks. The results of our first-use study suggest that Unravel addresses the design goals we formulated in Section 6.2.1. Participants found that Unravel provided transparency about the data (D1), allowed just-in-time exploration (D2), and minimized context switches between code and data (). In this section, we discuss our study results through the context of our design goals.

5.5.3.1 Visual Cues Helped Achieve Data Transparency

Participants relied on the visual cues to track transformations of a dataframe across the chain (D1). Summary boxes provided a useful visual cue for the basic properties of dataframe. Participants like P5, P6, P14, or P1 used the summary box dimensions to infer changes like adding columns or stripping rows from certain operations like `filter` or `summarise`. For example, P14 found that "it was very useful to have this at a glance information about the data shape and type of change at times because it supports quickly checking if the dataframes are correct." Upon discovering the row and column dimensions of the summary box, P5 thought "that feature of rows and columns numbers is I think one of the most powerful teaching things. It's really cool to see mutate is adding this column." The data change schema highlights were used by the participants to validate changes to the dataframe between steps. P4 expressed, "I like that you can flip between lines pretty quickly to see what changed, you have something that guides your attention. Being able to step through it and being able to walk through like look at this, look at that!" P5, a data science educator, commented how the internal change would be useful for teaching students about the behavior of grouped data: "I really like being able to show this internal change. This color scheme is really nice. Because I think students, even after you tell them they should expect it, they miss it."

5.5.3.2 Explorations on the Code and Data Enabled Checking Assumptions

Overall, we were able to achieve D2. Participants clicked on different parts of the chain, and explored the code using the drag-and-drop interaction and the toggle switches to validate their assumptions about the code and output. We found that being able to click on arbitrary lines of the fluent code was helpful for data scientists to inspect intermediate dataframes without being constrained to a linear stepping interaction like debuggers. The toggle switches to enable or disable lines helped participants explore the influence of certain functions when used with other functions. For example, P2 expressed that "it's cool how it helps dispel goofy assumptions about what attribute persists versus not. It made me examine so many assumptions especially grouping." Participants like P12, P8, P4 or P10 tested hypotheses about the code behavior by applying structural edits to the fluent code like disabling lines or reordering them. When examining the role of a `group_by` function for example, P10 guessed that "if you don't group by species then that would just work on the entire dataset." P10 then toggled off the `group_by` and confirmed "when you summarise the total now, it's applying this function across everyone in the dataset."

Participants explored and made use of the summary text to understand operations with large visible changes. Some R experts (P4, P2, P6) found that toggling lines on or off especially useful for understanding pivoting operations: "being able to quickly flip between lines after toggling things on or off is nice for these pivot_wider or longer functions." (P4). P2 found the summary text was useful for confirming their own summaries they made mentally: "I really like the pivot summary description." Sometimes, order effects were explored by participants. For example, P6 was able to validate the importance of placing a function like `filter` before running a summary function on the dataframe by reordering lines: "There are rows outside of mass that are also being dropped. This makes sense because I missed the hair_color so that's where I'm getting my counts of 28 versus 33 rows. This is interesting to be able to check your assumptions of getting the outputs."

5.5.4 Unravel Helped Minimize Context Switches Between Code and Output

Participants found the integration of Unravel into their IDE workflow to be useful. All participants commented that it was quite convenient that they could simply pipe (`%>%`) their own fluent code in R into a single function `unravel` to open up the tool in RStudio (D3). Upon unraveling a complex tidyverse code snippet P4 commented, "Oh wow, it actually worked! I like how you pipe it in at the end and it gives you this awesome thing." However, other participants wanted tighter correspondence between the text editor and the explorable code. For example, P3 expected the structural edits on the text editor to automatically update the code overlay and suggesting adding it as a feature since "it's so handy to not have to manually run from editor." Other participants (P4, P6, P9, P11) tried to edit the code in the explorable code itself, suggesting a need for live updates to the editor to further reduce context switches, especially if they want to copy the edited code.

5.6 Discussion and Future Work

In this section, we discuss the broad implications of our findings and identify the ways in which Unravel could be adapted to various programming languages and contexts.

Unraveling Code in Educational Settings. One common thread from our study was the excitement around using Unravel as an educational tool. Data science educators make use of computational notebooks, which students use to engage with content, code, and interactive elements like widgets or videos. These interactive documents enrich the learning experience by allowing learners to explore and understand code. P5 wanted to use Unravel as a teaching tool to author exercises in the spirit of Parson's Problems [Den08], asking students fix problematic fluent code using the structured drag-and-drop interactions. P9 commented that Unravel would have helped their tutoring sessions within the IDE because "it would've been really useful to walk through steps of how it starts getting data to how it ends." In a future version, Unravel could be used within interactive tutorials to explore code in RMarkdown [Sch21] or extended for Jupyter Notebooks.

Data Scientists Can Benefit From in-Situ Learning Tools. Instead of offloading to external learning resources, we should provide learning tools within the IDE or computational notebook. Data scientists can avoid context switches by using in-situ tools to generate code [Dro20; Ker20], version code [Ker17b], or manage messy code [Hea19]. Unravel builds on this approach by offering an *exploration mode* for fluent code within the IDE. An interesting application of Unravel could be to help data scientists understand code generated by programming by example techniques. Ferdowsifard et al. [Fer20]'s study of a live programming tool for python found that "programmers do not try to understand the code generated by the synthesizer." To encourage programmers to understand, tweak, and trust synthesized code, providing a means to explore code interactively with descriptions of operations might be useful. The data scientists in our study found it useful to use Unravel within the IDE because they could understand and explore their own code. However, some participants wanted live updates between the code in text editor and the explorable code overlay. In the future version, Unravel could incorporate elements of live programming techniques, syncing the structural edits to the original code. Based on our participants' focused behavior using Unravel, we believe that constant updates of dataframes during typing could be too distracting, so there should be a careful balance between liveliness and focused explorations.

Unravel in Other Programming Contexts and Environments. Unravel can be adapted to other programming languages and contexts to help understand and explore fluent code. Language-Integrated Query (LINQ) is a domain-specific language in C# that is used to query from various data sources such as relational databases (SQL). LINQ uses the fluent interface to filter on data:

```
List<int> numbers = new List<int>() {5, 4, 1, 3, 9, 8, 6, 7, 2, 0};
var orderingQuery = numbers
    .Where(num => num < 3 || num > 7)
    .OrderBy(n => n);
```

Structured explorations on LINQ queries can help C# programmers inspect and debug the chain

by allowing them to inspect each intermediate result, and explore variations with drag-and-drop or toggle interactions. Similarly, Python uses fluent code through method chaining to wrangle data using the `pandas` library:

```
df[['fl_date', 'tail_num', 'dep_time', 'dep_delay']]
    .dropna()
    .sort_values('dep_time')
```

Since each method returns a dataframe, we can analyse the code and split on method calls to store the intermediate dataframe for each operation (`[[`, `dropna`, and `sort_values`). Fluent code is also used in data processing frameworks like Spark, which is used for handling big data. Spark provides wrappers for many languages including Python and R. Programmers using Spark can use a tool like Unravel for interactively exploring fluent code that handles much larger data. Scaling Unravel to handle big data will require an effective way to summarise and visualize data transformations. Here, it might be useful to use Niederer et al. [Nie17]'s strategy behind TACO, an interactive comparison tool that visualizes the differences between multiple tables at various levels of detail. Instead of showing the entire table of each step, it might be useful to initially provide an overview of data changes in terms of row and column differences, before selecting a particular dataframe to get more details.

5.7 Conclusion

We explored the usefulness of Unravel, a tool that enables structured, drag-and-drop interaction with always-on visualizations to help data scientists explore and understand fluent code. Through examination of the R community, we identified several needs related to exploring fluent code. To address those needs, we designed Unravel which integrates within data scientists' IDE, helps them gain transparency about data, and explore fluent code using simple structural edits via drag-and-drop and toggle switch interactions. Through a first-use study with 14 data scientists, we found that Unravel facilitated diverse activities such as validating assumptions about the code or data, finding redundant or equivalent code, and learning about function behavior. Based on our results, we discussed some ways to generalize Unravel to other programming languages and contexts and identified future work to better support interactive exploration of fluent code.

CHAPTER

6

DEBUGGING DATA SCIENCE CODE

In the previous chapter, we studied how Unravel can help programmers learn data wrangling by exploring and understanding code through structured explorations, code and output highlights, automated function summaries, and interactive outputs. However, the study did not consider how Unravel might be used in a realistic scenario where a data scientist is in the process of data wrangling for the purpose of an analysis. Data wrangling requires data scientists to meticulously validate the data for potential problems, apply numerous transformations, rinse and repeat. The problem is exasperated when data scientists make mistakes during this process that either result in runtime errors or worse yet, silent data-related issues that break analysis downstream such as visualizations. There is a lack of research focused on helping data scientists understand, explore, and identify data-related issues such as missing elements, unexpected values, and bugs in the code.

To address this gap, we built extensions to Unravel that helps data scientists debug data wrangling code. We added a Data Details view that automatically displays descriptive statistics and potential data quality issues for all columns of a dataset and for each transformation step. To help data scientists better understand code behavior, we also added a Function Help feature that allows users to click on functions on the Code Overlay to open the documentation within the IDE. In a user study with 18 data scientists, we found Unravel helped data scientists explore data wrangling code, triangulate and validate many assumptions about problems in the code or data for each transformation through the code, interactive outputs, and data details. We discuss the design implications of the future interactive exploration tools for data science programming.

6.1 Motivation

Data scientists spend a significant amount of effort learning, using, and debugging data transformations to prepare the data for analysis. Data wrangling is a tedious and error-prone process because it involves iterating on numerous data transformations so that it's ready for a given analysis task [Ker19]. In machine learning, data scientists have to iterate on different model features, architectures, and hyperparameters [Ame19]. One key characteristic of data wrangling code is that it involves "exploratory" programming [Ker17a], producing messy code that can introduce data-related issues that go unnoticed until an analysis is performed. Currently, there are tools for organizing the messy code such as Gather [Hea19] or comparing versions of code like Verdant [Ker19]. However, it is also important to understand and validate those changes that result from the data scientist's transformations. This is made difficult when data already comes with issues such as missing values, or outliers that affect analysis downstream. To deal with these issues, data scientists are forced to write additional code to perform these checks in a data wrangling pipeline. In other words, data scientists have to engage in "yak shaving", what you are doing when you're doing some stupid, fiddly little task that bears no obvious relationship to what you're supposed to be working on." [Bro00]

Recent research has studied ways to help data scientists explore and comprehend data transformations. For example, Datamations uses animations to visualize and explain common data transformation pipelines in R [Pu21], and this idea has been further explored in web-based tools that visualize how a table is transformed from one step to the next with Tidy Data Tutor [SK22] for R, and Pandas Data Tutor for Python [SL22]. There are also tools such as Data Wrangler [Kan11; Hee21] that help compose transformations by using GUI-based interactions to preview and execute transformations reducing the effort to code. While these tools by themselves are useful for exploring and understanding code, the tools do not remove the manual validations required to ensure data is being transformed properly. Data scientists have also expressed the need to integrate data exploration tools within their familiar working environment like Jupyter Notebook [Dro20]. However, there is a lack of research around tooling within a larger programming environment such as Integrated Development Environments (IDEs) to support them in learning about, and debugging data wrangling code for exploratory analysis.

We address the issues related to performing manual validations on data, understanding data wrangling code, and integrating learning tools within data scientists' workflows with Unravel [Shr21b]—an in-situ tool for the RStudio IDE designed for data scientists to explore, understand, and debug data wrangling code and output. Shrestha et al. [Shr21b] found that structured explorations of data wrangling code and output is useful when paired with always-on visualizations that provide at-a-glance information about the changes that have occurred for each step (e.g. data shape). The authors also found that highlighting correspondences between the code and the output helped data scientists validate assumptions about code behavior and expectations about the data. However, it is unclear how well Unravel can support data scientists during exploratory analysis tasks, especially with regards to both identifying potential data smells [Sho22] and helping them identify mistakes

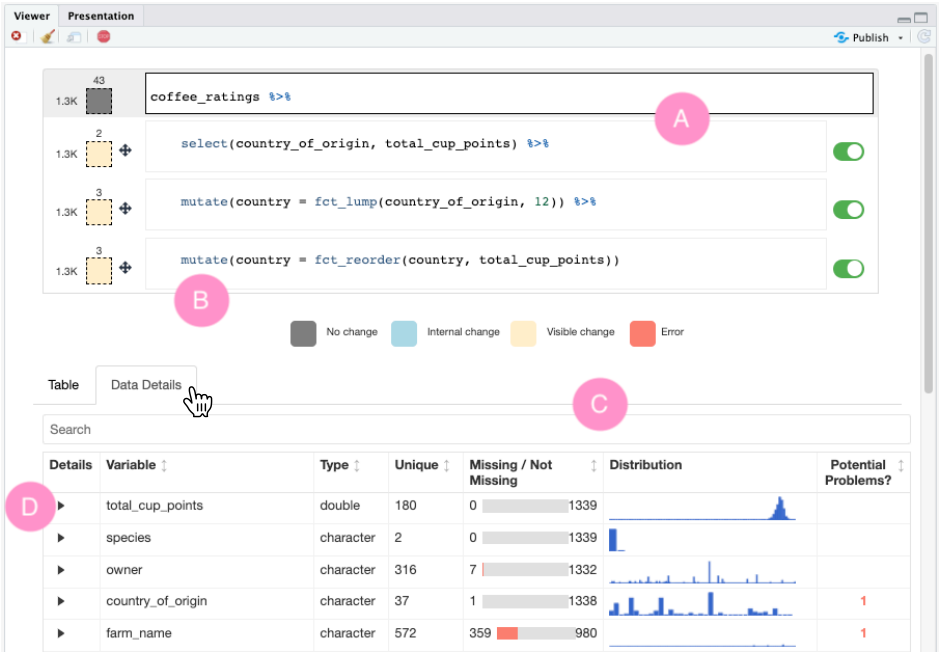

Figure 6.1 Data scientists can use Unravel to explore, understand and debug data wrangling code and data. Users can unravel code and interactively explore it on the Code Overlay (A). Users can click on hyperlinked functions to open the Help documentation page for the function (B). The Data Details tab displays an overview of each column of the dataframe at a particular line with the column (variable) name, its type, the number of unique elements, a missing versus not missing bar, a histogram and potential problems (C). To examine more details for each column, the user can then click on the carat icon to display more statistics and potential issues (D).

they make while writing data wrangling code. Wang et al. [Wan22] explored an interaction model called Diff In The Loop (DITL), which emphasizes providing data and distribution differences for versions of data wrangling code in Python. While data diffs are important to convey, our formative interviews with 8 data scientists revealed that there is also a significant amount of time spent verifying data quality issues (e.g. missing values), and they find that exploration of data wrangling code is difficult because they have to manually edit code to track changes, or selectively execute code to isolate and validate particular outputs for issues.

Based on our insights from the formative interviews, we built extensions to Unravel in order to reduce the manual efforts of validating code behavior and transformations, as well as identify potential data quality issues. Unravel eases code exploration by turning code into an interactive overlay within the IDE and structured editing of a data wrangling code. We extended Unravel with *automated data quality checks* in the form a "Data Details" view, which displays descriptive statistics about particular types of columns (e.g. numeric versus categorical), inline histograms as always-on visualizations to display these statistics. We conducted a user study evaluating these design features and found that Unravel and the Data Details view supported data scientists in both exploring code

and data, as well as debugging data quality issues and mistakes in the code.

The contributions of this paper are the following:

1. An in-situ interactive tool called Unravel that we demonstrate reduces data scientists' data wrangling efforts by providing structured explorations of data wrangling code and automated data quality checks for the code and output at each data transformation step.

2. Through a user study with 18 data scientists, we provide insights on the design tradeoffs of Unravel for debugging data issues, and point towards future work to improve ways we can help reduce friction to data wrangling programming.

6.2 Related Work

Data science tools to manage code and outputs. Researchers have developed tools that help data scientists write and modify code. Kery & Myers [Ker17a] have found that data science programming is characterized by exploratory programming, which can lead to disorganized and ephemeral code. Tools like Gather [Hea19] help analysts find, clean, recover, and compare versions of code in cluttered, inconsistent computatl notebooks like Jupyter. To explore alternative code in notebooks, Fork It [Wei21] uses a technique to fork a notebook and directly navigate through decision points in a single notebook. We designed a just-in-time learning tool to explore code by allowing temporary exploration through interactive code overlays and structural edits on the code itself, for example, by enabling, disabling or reordering lines. To help data scientists automate writing data wrangling code, Wrex [Dro20] uses programming-by-example. Similarly, mage [Ker20] is a tool that helps users generate code based on the modifications made from interacting with dataframes. Unravel builds on these by helping data scientists understand and iterate on the human or machine-synthesized code.

Understanding data science programming. Prior work has explored some tools to help data scientists understand code. For example, Lau et al. [Lau21]'s TweakIt is a system designed to help end-user programmers collect, understand, and tweak Python code within a spreadsheet environment. There are also tools that help data scientists visualize how common data wrangling operations work. Pu et al. [Pu21]'s Datamations tool animates dataframe wrangling and visualization pipelines in R.Datamations automatically processes fluent code in R using `tidyverse` [Tida] packages and provides a paired explanation and visualization of each step in the chain. Unravel is designed as an interactive tool used within RStudio IDE to allow opportunistic learning and debugging of data wrangling code. There are also web-based tools like Tidy Data Tutor that visualizes functions in R to help data scientists visualize how those functions transform data [SK22]. `pipediff` [Fab22] is an RStudio plugin that highlights the differences between two adjacent steps in a data transformation pipeline within the IDE. For data quality checking there are fewer tools available.

Debugging data wrangling code and data. More recently, there has been some focus on data quality issues and tooling to help fix them. Diff In The Loop (DITL), emphasizes the idea of displaying

data and distribution differences for versions of data wrangling code [Wan22] during exploratory analysis work. Our work is most closely aligned with DITL with regards to problem motivation and some aspects of the design. While Wang et al. [Wan22] uses the idea of code snapshot differences, we focus on transformation differences in a data wrangling pipeline for one version of the code at a time. The DITL prototype also shows various descriptive statistics and visualizations like histograms and their differences between these code snapshots, which can be useful for debugging transformations. Our approach shows distributions using in-line histograms as an always-on visualization for each step, and we track changes between transformation operations not code versions. We complement this work by also focusing on data quality issues as yet another primary concern in the data wrangling process. There is also a web-based tool called Rill Developer [RD22] that automates and updates data quality checks for SQL tables while the user types queries. Our Data Details view automates checks, but we do not implement live programming for the sake of simplicity and reducing potentially distracting updates while typing.

There has also been some work on validating specific types of input that is common to both programming in general and specific to data science programming. Programmers might omit input validation since the inputs can appear in many different formats. For example, a string can be represented in many different ways that describe domain-specific formats such as dates ("2022-06-02"). Scaffidi et al. [Sca08] explored the idea of enabling programmers to validate these inputs by using the idea of a "tope": an application-independent abstraction describing how to recognize and transform values in a category of data. The authors created an interactive application that allows one to specify the format of a string for example, which the system can use to flag valid versus invalid inputs. This was found to help improve the accuracy and reusability of validation code and facilitates data cleaning such as duplicate identification. Recent work in program synthesis and programming-by-example also builds on this idea by making it easier for programmers to teach a system to learn how to transform certain types of inputs for refactoring [Ni21] or identifying regex input patterns [Zha20b]. While our work is not specifically designed to form and identify patterns of input types, we use type-specific validation to display potential problems that are specific to the data type. For example, warning programmers about extreme values in numericals or miscoded `NAs` in string values such as empty spaces. The "topes" idea, Ni et al. [Ni21]'s mixed-initiative technique to identify and valid common transformation, and Zhang et al. [Zha20b]'s interactive programming-by-example technique to form regexes could further improve our system by allowing a user to be more specific about input formats for robust validation.

6.2.1 Formative Interviews and Design Goals

To better understand the common pain points of exploring data wrangling code and the types of data quality issues that are commonly dealt with, we interviewed eight data scientists who frequently use the RStudio IDE to wrangle data in R. In our interviews, we focused on how they perform data wrangling, how data wrangling fits within their IDE workflow, what tools they use or have used for checking data quality issues, and what difficulties they face as they wrangle data. These data

scientists (F1–F8) provided several insights that guided the design goals for Unravel and the new Data Details view to diagnose issues with the code or data.

For exploration of data wrangling code and output, data scientists expressed that current tools make it difficult to navigate the many steps in a transformation pipeline, and understanding how the code and output corresponds with each other. All of the data scientists made heavy use of features within RStudio that helps them explore data. F4, F5, and F8 talked about how they typically start by previewing the first or last few rows of the data and the column or variable types. However, this is typically insufficient since console outputs in IDEs like RStudio "may not reveal potential issues with data like missing values. " (F4) Similarly, F1, F2, and F3 talked about how they also have difficulty "catching subtle issues" (F1) like specific values that one has to normally write code to filter out. This process is made difficult when there isn't an interactive way to explore a data table, with a few data scientists (F2, F4, F8) expressing that while they do make use of interactive tables in RStudio using `View` function, it can be hard to manage all of these ad-hoc tables in the IDE. Finally, data scientists had trouble isolating individual transformation steps to understand and validate changes that they have made to the data. These issues altogether lead to multiple views and tools used for exploratory checks that can take a data scientist out of context from their data wrangling code. Thus, our first design goal is the following:

> **D1:** Data wrangling tools should make it easy to navigate and summarize the outputs of transformation steps in a consolidated view that reduces context switching.

Data scientists expressed that a significant portion of their time is spent on checking data quality issues, and validating each transformation of the data. When exploring data for the first time, all data scientists described common checks that they perform manually by writing code such as checking for variable types (e.g. string versus a number), missing values (`NA`s), and miscoded [Gre19] `NA`s values (e.g. -99, "-", etc.), outliers, unexpected values, distribution of variables, and descriptive statistics (e.g. ranges). F3—F5 described using a package in R called `skimr` [War22] which summarizes these checks and descriptive statistics into a text output, but this still didn't allow "interactive introspection into each of these characteristics" (F3). F1, F4, F5, and F6 found histograms of variables in particular is quick and easy way to understand the characteristics of the dataset as a whole. This leads us to our second design goal:

> **D2:** Data wrangling tools should help data scientists find mistakes in their code and include automated descriptive statistics and checks after every transformation.

6.3 Design and Implementation

We present the implementation of the extensions we added to Unravel: Code Overlay enhancements, Function Help, and the Data Details. The other existing features and their design has been discussed in [Shr21b]. We discuss our design decisions for these extension features to support our design goals in Section 6.2.1.

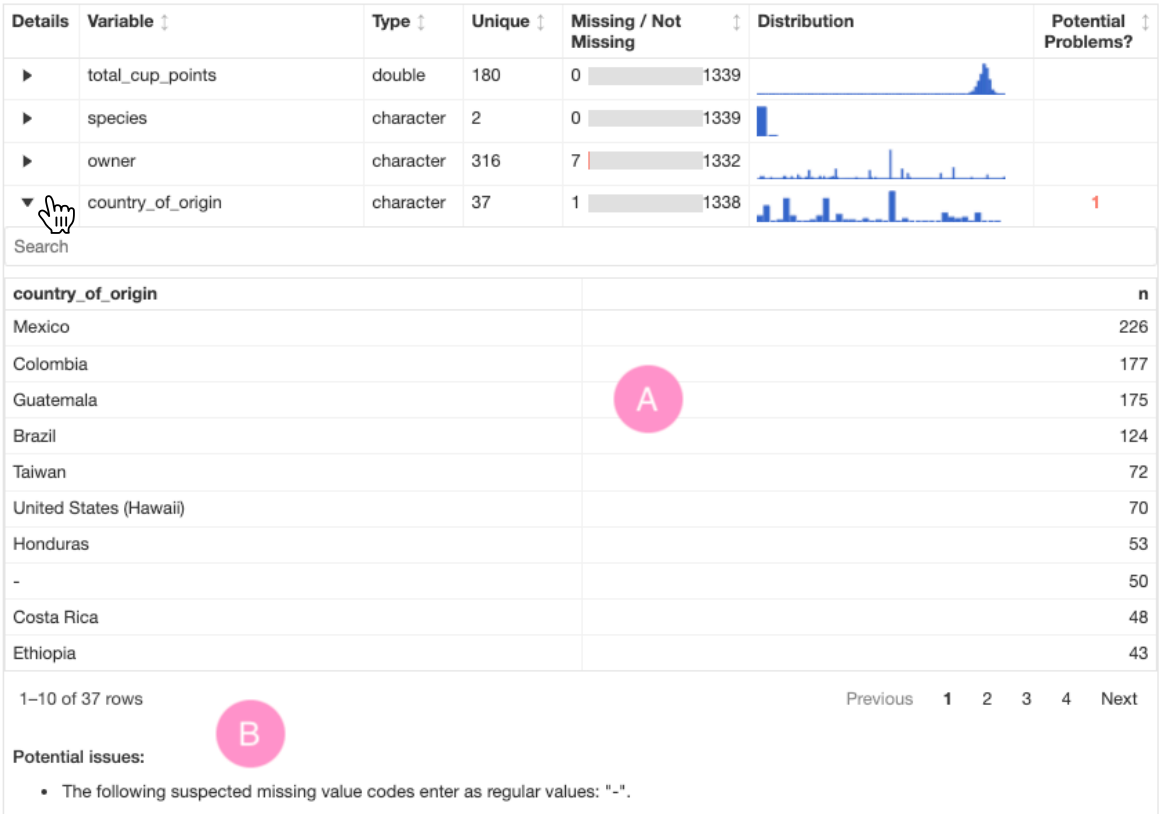

Figure 6.2 A user can examine extra details about the column for the dataframe at each line displaying a type-specific statistic such as a count table (A), and potential issues (B) for the type such as miscoded `NA`s like "-" for categorical columns.

6.3.1 Exploration Mechanics

We added a couple of enhancements to Unravel to make it easier to explore and debug data wrangling code in R. The previous study on Unravel did not explore how the broken transformation steps and their associated error messages through the function summaries could help data scientists debug data wrangling code. We decided to explore this to support D2 with respect to finding mistakes in data wrangling code by exploring how the Error change type color can act as an always-on visualization providing an immediate visual cue that the pipeline is broken. We implemented an *error-forward* strategy when dealing with errors in the transformation pipeline where we still render code that results in an error so that data scientists can still explore and understand the code. They can also flip off lines that have errors and read the error message through the function summary feature to understand the root cause of the broken data wrangling pipeline. This supports debugging code-related errors that can occur by rendering up to the point of the problematic line. If a line causes other subsequent lines to fail, Unravel will display a message through the function summaries that "Previous lines have problems." Data scientists can then work up to the first line that broke and

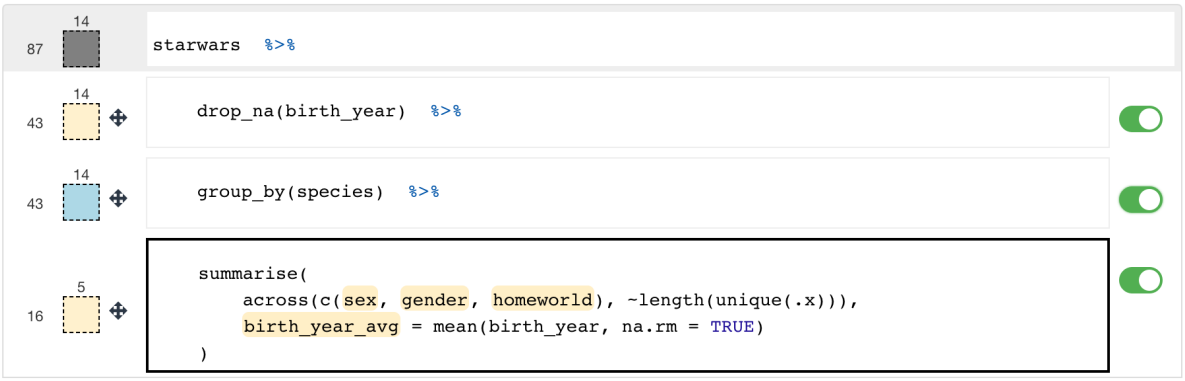

Figure 6.3 The columns referenced in the Code Overlay can now be referenced in arbitrarily nested expressions, making sure to highlight the changed or new columns.

read the error message to fix their issue.

6.3.2 Code Overlay

One central feature of Unravel is the Code Overlay which takes data wrangling code written in R and makes it interactive for ease of exploration and debugging. A limitation of the Code Overlay was that the code highlights did not render for arbitrarily nested expressions that contained variables [Shr21b]. The authors reported limitations regarding the code highlighting due to the complexity of program analysis. One challenge is that R allows functions' parameter values to be function calls themselves. This means that for almost all functions in R, one could include column references that are inside arbitrarily nested function calls as the value of function parameters. The authors scoped Unravel to initially handle the simpler parameter values like the example below. Another challenge is to disambiguate between a column with the same name as a function making sure only to highlight the column.

We extended Unravel to solve these challenges by analyzing the AST of data wrangling code further. We make use of the R function `getParseData()` to assist in the analysis of the AST by storing the parse tree of each line's code. Symbols that correspond to column names are stored which are used to identify the text to highlight in the Code Overlay and to differentiate them from function calls. This helped us solve the challenge of finding column references in arbitrarily nested expressions. We also improved the way code highlights work by using pattern matching on the code text such that function calls inside functions like 'summarize()' are excluded when highlighting columns. This solves the challenge of differentiating columns from function calls and therefore highlights relevant columns in arbitrarily nested calls (Figure 6.3).

6.3.3 Function Help

To help achieve D1, and partially D2 (Section 6.2.1), we added the idea of hyperlinked functions on the Code Overlay that would open up the help documentation page for that function on RStudio's

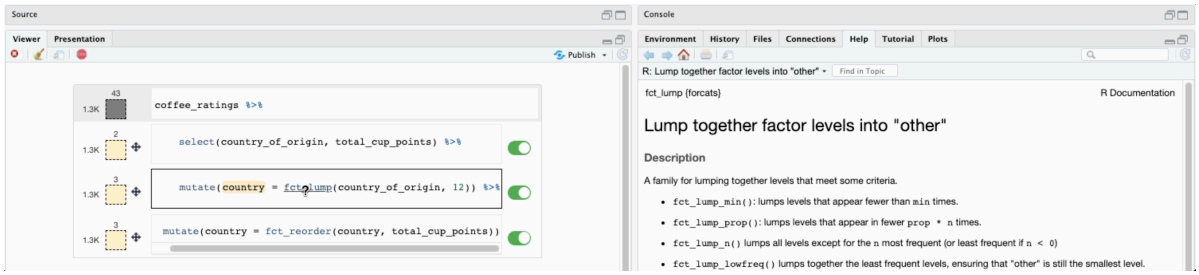

Figure 6.4 A user can click on a hyperlinked function to open its documentation in the Help pane.

Help pane. Programming in R involves a lot of functional programming, and the documentation is usually geared towards how to use these functions which can be access through a dedicated Help pane. To achieve D1, we decided to use this Help pane to open documentation for functions whenever a data scientists uses Unravel and is confused with the functions used. We also open the Help pane besides the Viewer pane where Unravel is rendered so that data scientists can read the documentation alongside the tool.

The process of forming and rendering these hyperlinked functions on the Code Overaly involves a similar to the process behind collecting column references to highlight in the code text and output (Section 6.3.2). We make use of the `getParseData` function to extract the AST for the code at each line. We then analyze the AST and store the functions used in a particular line and store metadata about the function name and which package it belongs to. The metadata is extracted using the namespace search mechanism in R that helped us find the package that the function belongs to. This information is used on the frontend to wrap a function call text such as `group_by` with a hyperlink. When the hyperlink for the function is clicked, we request the server to invoke `help(<function>, <package>)` function to open up the documentation on the Help pane in RStudio.

6.3.4 Interactive Tables

To achieve both D1 and D2, we enhanced the interactive tables by adding better clues on grouped dataframes, column types, and search by value.

We enhanced features related to conveying subtle changes and important variable information. One of the common patterns in data wrangling is grouping the data according to certain variables and then doing an aggregate computation such as finding the average for each group. While Unravel does implement color change schema to visualize the types of changes occurred, grouped dataframes require some additional information such as the number of groups. We added this information on the interactive table that displays the dataframes whenever there is a `group_by` operation (Figure 6.6). In addition, we also added the type for each column of the table like `<int>` for integer or `<chr>` for character, mimicing the console output format for dataframes when using the tidyverse R packages.

Data scientists in our formative interview mentioned that finding erroneous values that don't meet their expectations are often easy to miss (Section 6.2.1). For example, miscoded `NA`s are

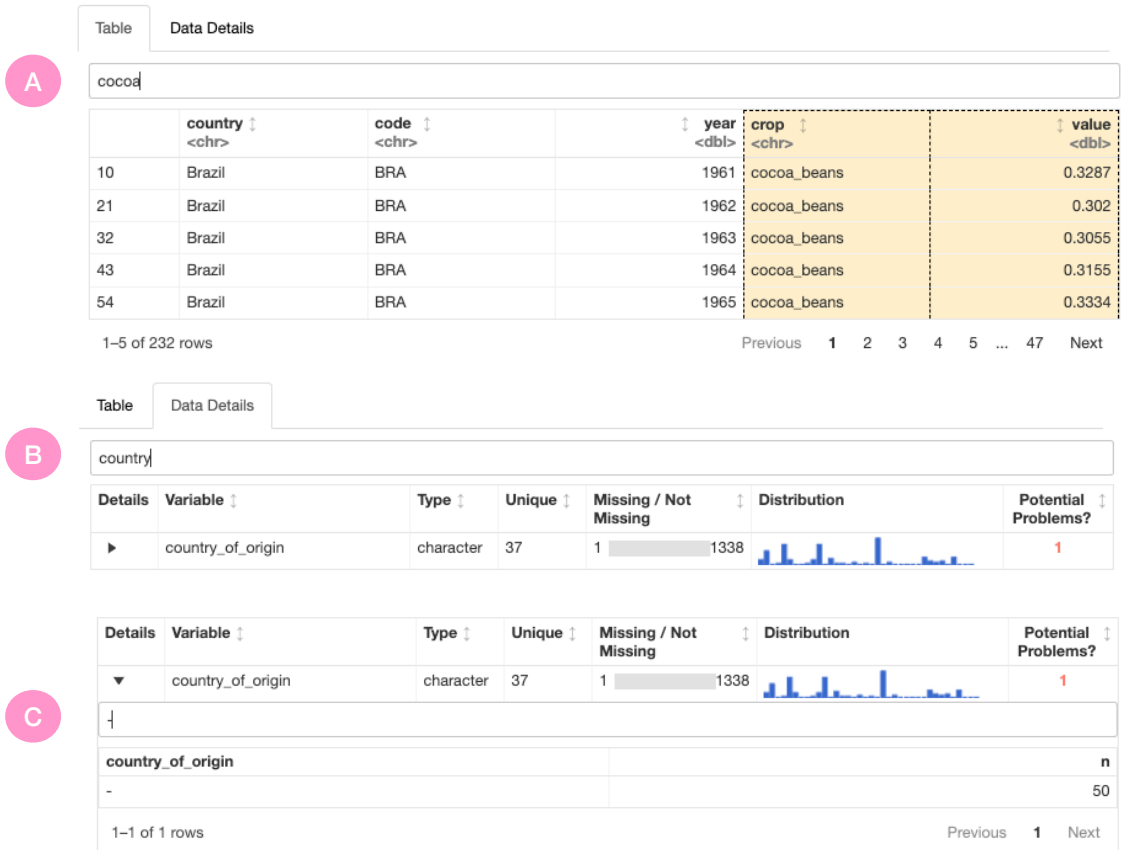

Figure 6.5 Users can search for particular cell values within the Table output (A), relevant columns within the Data Details table (B), and expanded detail tables for a column which is useful for categorical variables (C).

problematic because they can be used as a placeholder for missing values (e.g. -99) yet go undetected using conventional approach of searching for `NA`s since they are numbers. To facilitate this process, we added search bars (Figure 6.5) for the interactive table so that data scientists can search for specific values that will be matched via pattern matching. This same functionality is added for the Data Details tables discussed in the next section.

6.3.5 Data Details

To achieve D2, we implemented a new view that we call Data Details (Figure 6.1, Figure 6.2). In our formative interviews, several participants were expressing the need for quick summary statistics on all of the variables of a dataframe. They mentioned using built-in functions in R such as `is.na` or packages like `skimr` [War22] which provides a static printout of the column types, their descriptive statistics (for e.g. IQR), number of missing values, and a basic histogram showing the distribution of the column. Along with automating the summary of variable characteristics, data scientists also desired reducing the manual writing effort to perform data quality checks that can be present for

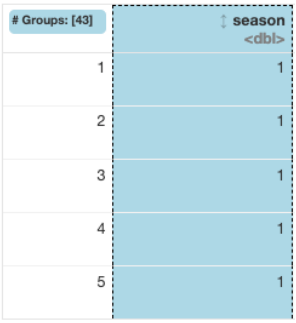

Figure 6.6 Users can take a glance at the number of groups for a grouping variable.

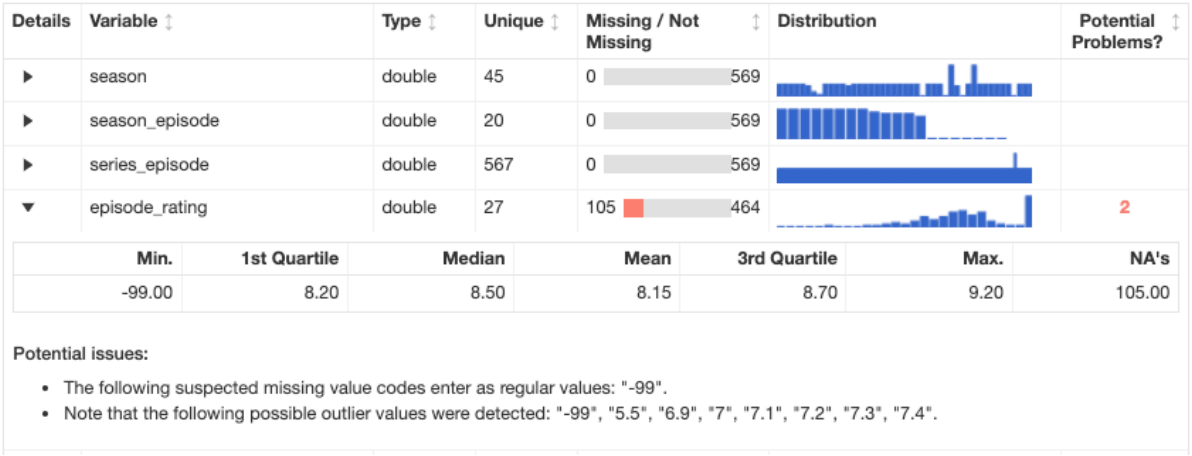

Figure 6.7 Users can examine summary statistics about a particular column when expanding its row, and be aware of potential issues This an example of details for a numeric type.

any type of dataset such as missing values or `NA`s, miscoded `NA`s, outliers, and unexpected values.

We implemented a Data Details interactive table next to the interactive output table for each transformation step in the code that displays the type, number of unique elements, number of missing versus not missing elements, the distribution, as well as the potential problems in the column. Whenever a data scientist wishes to view the details behind a particular dataframe in a transformation pipeline, they can click on the "Data Details" tab. They are then presented with an interactive table that contains on the high level descriptive statistics. We used existing built-in functions like `summary` for numeric variables (Figure 6.7) and functions like `count` from the `dplyr` package for counting values in categorical variables. Each column allows further exploration of these details describing the data type (numerical versus categorical) in more detail and the potential issues they have to fix.

6.4 User Study

We conducted a 60-minute long usability study over video conference with data scientists to understand how Unravel can assist them in understanding and debugging exploratory data wrangling

code for exploratory analysis. We investigated the following research questions:

- **RQ1:** Does Unravel help data scientists explore and understand data wrangling code for exploratory analysis?
- **RQ2:** How does Unravel help them identify data quality issues and debug data wrangling code?

6.5 Method

6.5.1 Recruitment

We recruited participants (Table 6.1) through an online advertisement of the study on Twitter where data scientists are increasingly active [Moc22]. To be eligible for the study, participants had to self report basic experience with R programming. We recruited 18 data scientists (10 males, 8 females) who had varied levels of experience in data wrangling and programming in R. On a 5-point Likert scale, participants self-reported their experience in data wrangling ($\mu = 3$) and R ($\mu = 3.1$), with a minimum of 1 and maximum of 5. Participants worked in software engineering (3), research (3), statistics (2), education (2), genetics (2), psychology (2), data science (1), social work (1), marketing (1), and IT (1).

6.5.2 Study Setup

The study was conducted remotely with participants sharing their screens over a video conferencing tool. For the programming environment, we used RStudio Cloud IDE, a web-based version of the RStudio IDE. Since this is a browser-based IDE, participants were able to use the tool on their computers within their own choice of browsers and configurations.

Study Phases: Each study consisted of three sessions: a demo session and three debugging tasks. For the demo session of Unravel, we showed participants how to use the tool with some basic code snippets. The demo task covered the main components of Unravel: code overlay to explore the different lines of the code, the shape and summary boxes always-on visualizations, the interactive data table, and the Data Details. We nudged participants if they were stuck or confused about how to use the tool. After the completion of the study, we administered an exit survey to measure the usefulness of Unraveland to ask for additional feedback from participants.

Instrumentation: We instrumented the Unravel and RStudio to log all successful user code executions and Unravel-specific feature usage. For Unravel, we logged several events for unraveling code, clicking on the intermediate lines in the code overlay, clicking on function summary box, clicking on hyperlinked functions to open documentation, clicking the toggles, and reordering lines. To better understand how the interactive table and the Data Details were being used, we also logged the counts of "focusing" on the tables (via mouse hover event). To analyze all of this data, we computed the count of logged entry by type, and report the counts, average and standard deviation across participants.

6.5.3 Debugging Tasks

For the three debugging tasks, the participants were tasked with exploring analysis scripts written in the tidyverse [Tida] R using the `dplyr` and `tidyr` packages with the goal of fixing the data wrangling code to produce a visualization. The three datasets were based on #TidyTuesday datasets [Moc22], which included data quality issues by themselves, and we further modified them to include common data smells found in prior work [Sho22] and the issues mentioned by participants in our formative interviews.

Each script was scaffolded with code to import the dataset, wrangle the data, and visualize it to explore relationships between variables. Participants were tasked with thinking aloud while understanding, exploring, and debugging the data wrangling code using Unravel. We did not prevent participants from writing their own code to explore and validate the dataset using RStudio. This was done to investigate when participants would reach for Unravel rather than writing their code. The tasks were designed to examine how data scientists of varying experience would discover and debug issues in common types of data wrangling operations like selecting, filtering, mutating, grouping, and summarising dataframes. In particular, we designed 3 tasks varying in both wrangling operations and types of data quality issues reported:

A. Coffee Ratings: In this task, participants examined a script that visualizes total coffee ratings for countries around the world using a boxplot.
Problems: The data wrangling code involved filtering out missing values (`NAs`) or miscoded `NAs` like " " (empty space) or 999. The missing values are not handled in data wrangling code which produces an incorrect plot containing boxplots for total cup points by top 12 countries. The inclusion of miscoded `NAs` makes this process harder because they are no longer detected with explicit checks for `NAs`.

B. Chopped: For this task, participants examined a script visualizing the average episode ratings for all seasons of the Chopped TV Show.
Problems: The data wrangling code included an incorrect order of summarizing statistics about particular groups of variables before grouping the data by those variables. Grouped calculations is a common programming pattern in data wrangling yet it can be confusing because it collapses large tables into a smaller one by aggregating values according to groups. The data also contained a problematic value (-99) that fell outside of the range of expected values for a rating (0-100), which affects the visualization by producing a very low average rating for season 9.

C. Crop Yields: Participants examined a script that analyzed the global crop yields for major countries like USA, Brazil, China, and Russia.
Problems: The data wrangling code reshaped the dataframe from a wide form (many columns) to a long form (many rows). However, the existing code using `pivot_longer` is used incorrectly because it does not produce the right column name for the crop yields value. The code

included another mistake where the `year` column was converted from a `numeric` type to a `character` (string).

Participant	Gender	Field	Task A	Task B	Task C	Wrangling Exp.	R Exp.	Skill Level
P1	M	Software Engineering	✓	✓	✓	3	2	Beginner
P2	M	Software Engineering	✓	✓	✓	4	3	Experienced
P3	M	Software Engineering	✓	✓	✓	4	2	Beginner
P4	M	Organizational Pyschology	✓	✓	○	4	3	Experienced
P5	M	Education	✓	✓	✓	5	4	Experienced
P6	M	Education Technology	○	✓	○	4	4	Experienced
P7	M	Statistics Education	○	✓	○	3	3	Beginner
P8	F	Cancer and Genetics	✓	✓	○	3	4	Experienced
P9	F	Pyschology	○	✓	○	2	3	Beginner
P10	M	Marketing	○	✓	✓	3	3	Beginner
P11	F	Statistics	✓	✓	○	3	2	Beginner
P12	F	Genetics	✓	✓	✓	4	4	Experienced
P13	F	Social Work	✓	○	○	3	3	Beginner
P14	M	Ecology	✓	✓	✓	4	4	Experienced
P15	F	Pyschology	✓	○	○	2	3	Beginner
P16	F	Public Policy	✓	✓	✓	3	4	Experienced
P17	F	Data Science	✓	✓	✓	3	3	Beginner
P18	M	IT Enterprise	✓	✓	✓	4	4	Experienced

Table 6.1 Demographic information and task completion results. The cells are marked with a ✓ to indicate they successfully completed the tasks (Section 6.5.3) by fixing all the code and data mistakes. The ○ indicates that they were not able to start or complete the task. Wrangling and R Exp. are the participants' self-ratings for data wrangling and R experience using a likert scale of 1-5 (Novice to Expert). We used the average of data wrangling and R skill ratings to bucket beginners (2–3.5) and experienced (above 3.5) to facilitate analysis of the user study.

For each task, we included inherent issues with the data as well as mistakes in the data wrangling code to understand whether Unravel can help before analysis. To observe whether and how participants reached for Unravel or performed their own checks, we did not prohibit participants from using whatever code they felt necessary to write to understand and fix the issues with the data wrangling code. Further, this helps in determining in what situations Unravel may or may not have been helpful.

6.5.4 Analysis

We used a mixed-methods approach to analysing the results of our study through the two research questions:

RQ1: *Does Unravel help data scientists explore and understand data wrangling code for exploratory analysis?* To answer RQ1, we conducted qualitative analysis by first taking notes according to which features were used and in what way to facilitate exploration and understanding. We then

Table 6.2 Post-Study Survey Responses

		Likert Resp. Counts[1]					
	% Agree	SD	D	N	A	SA	Distribution[2]
Clicking on functions to view its documentation was useful.	100%	0	0	0	5	13	
The Data Details view helped identify potential issues with data.	94%	0	0	1	4	13	
Clicking on lines to view intermediate data was useful.	94%	0	0	1	5	12	
The data change color schema helped describe transformations.	89%	0	2	0	2	14	
The Data Details view helped describe transformations.	89%	0	2	0	5	11	
The dataframe shape information of row and columns helped validate changes.	89%	0	1	1	4	12	
The tool was useful overall.	89%	0	0	2	5	11	
Invoking Unravel through code highlight and Add-in was useful.	83%	0	0	3	9	6	
The summaries helped describe the effect of functions.	83%	0	1	2	3	11	
The toggle switches to enable/disable lines helped explore the code.	83%	0	1	4	5	7	
The ability to reorder lines helped explore the code.	72%	0	1	4	4	9	

[1] Likert responses: Strongly Disagree (SD), Disagree (D), Neutral (N), Agree (A), Strongly Agree (SA).
[2] Net stacked distribution removes the Neutral option and shows the skew between positive (more useful) and negative (less useful) responses.
Strongly Disagree, Disagree, Agree; Strongly Agree.

performed open coding on our notes to look for initial patterns, and related the codes to form higher level themes that describe how the Unravel features helped the participants. To triangulate our findings, we analyzed the log events (see Section 6.5.2) for patterns that help explain these themes.

RQ2: *How does Unravel help them identify data quality issues and debug data wrangling code?* To answer RQ2, we conducted qualitative analysis by examining when and how participants used Unravel's Data Details view to identify data quality issues, and debug the data wrangling code mistakes. Similar to RQ1, we conducted qualitative analysis on notes taken according to the usage patterns we observed during the study. Then, we analyzed the logs to triangulate and support our qualitative findings.

6.6 Results

We present both of our qualitative and log analysis results from the user study, describing the behaviors we observed, the strategies used, and the feedback participants provided. The results of our user study suggest that Unravel addresses the design goals we formulated in Section 6.2.1. Participants found that Unravel provided helped them explore data wrangling code when they felt confused about transformations (D1), and used Data Details to further understand and most importantly catch issues with both the code and the data (D2). In this section, we discuss our study results through the context of our design goals.

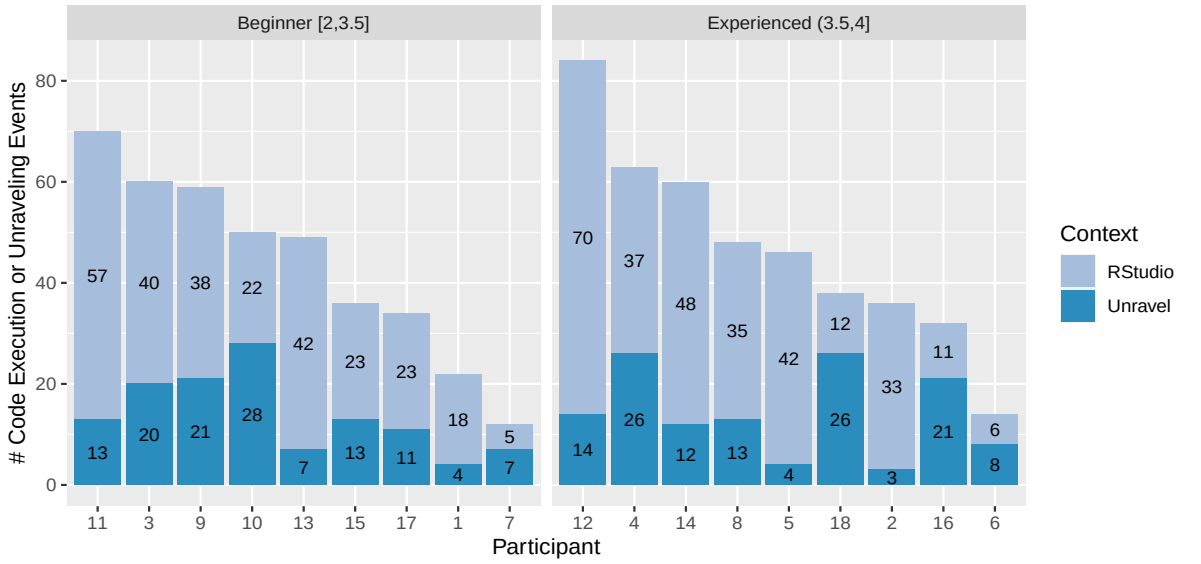

Figure 6.8 Number of times participants executed code in RStudio or used Unravel to explore code.

6.6.1 Post-study Survey Results

On a 5-point Likert scale (Table 6.2), participants positively rated the usefulness of Unravel overall ($\mu = 4.5$). Participants rated the function click to help documentation ($\mu = 4.7$), clickable lines for viewing intermediate dataframes ($\mu = 4.6$), the change color schema ($\mu = 4.6$), Data Details to catch potential issues ($\mu = 4.7$) and understand transformations ($\mu = 4.5$), and the always-on visualization of the dataframe shape ($\mu = 4.5$) as the most useful features. Other features were lower in rating such as function summaries ($\mu = 4.4$), drag and drop to reorder lines ($\mu = 4.2$), invoking Unravel by highlighting code and using it through the RStudio Addins ($\mu = 4.2$), and toggle switches for enabling or disabling lines ($\mu = 4.1$). Some of the existing features were rated similarly in the first-use study [Shr21b], and the new extensions such as function click to help documentation and Data Details were thought to be useful features.

6.6.2 RQ1: Does Unravel help data scientists explore and understand data wrangling code for exploratory analysis?

Interactively exploring data wrangling code and output: Overall, Unravel made exploration of data wrangling code and output easier because "it offers a lot of information in a very compact way." (P14) Several features of Unravel helped participants interactively explore the code behavior and the dataframe transformations (D1).

As shown in Figure 6.8, Unravel was used in bursts whenever participants were trying to understand code behavior or the resulting dataframe from the wrangling. Beginners and experienced participants did not differ much on their frequency of using Unravel or executing code in RStudio. Beginners unraveled code 124 times, and executed code 268 times in RStudio while experienced

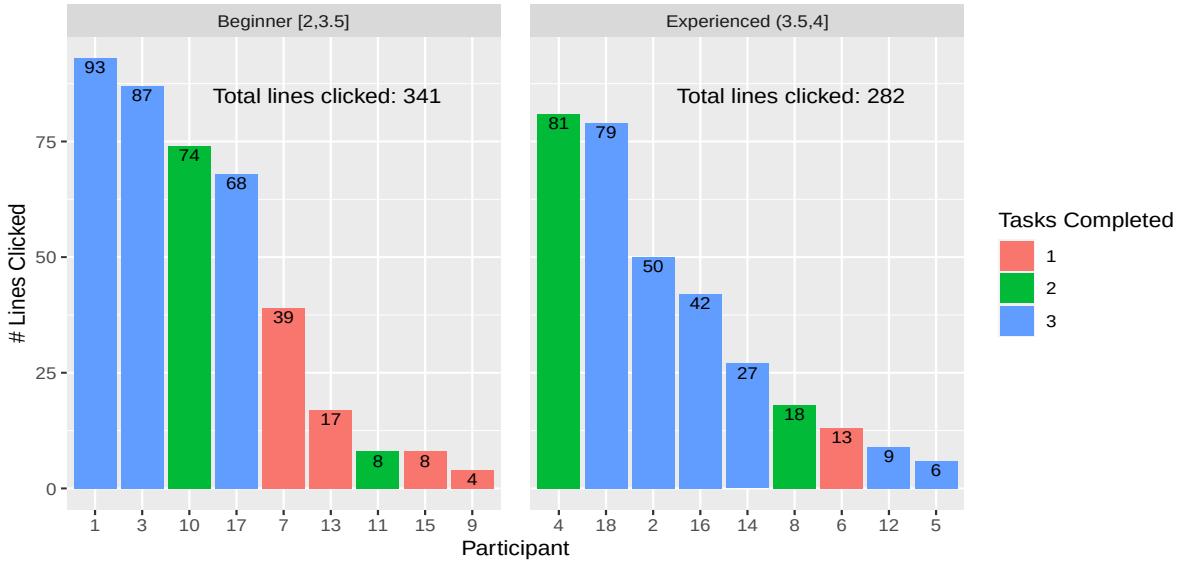

Figure 6.9 Number of times participants clicked on the intermediate lines in the Code Overlay.

participants unraveled code 127 times and executed code 294 times. We observed a mix of beginners and experienced participants making use of RStudio features during their examining of data, such as the Environments pane (P5, P7), the interactive table using the `View()` function (P5, P6, P9, P10, P18) that holds all variables such as the dataset variable, and a few used the interactive console (P4, P5, P10).

When participants were using these other views, they eventually decided to use Unravel when they needed to view both the code and the output in the same window. When trying to isolate lines, some participants (P4, P8, P9, P10, P12, P13) sometimes used Toggle Switches (57 times). For task B, many participants (P1, P3, P4, P7, P9, P10, P11, P14, P16–18) used the Drag to Reorder feature (33 times) to explore and fix the incorrect order of the existing code where `group_by` step is placed after `summarize`. P13 favored the Unravel's interactive tables when checking dataframe outputs because of the ease of exploring them in the tool: "I'm a very slow coder, I change things and come back and mess things up each time. [In Unravel], I can see that I'm messing up if my dataset has a strange shape and not over on the console. I check the console output but it's not as easy to understand as what happened [in Unravel]." Experienced participants like P5, P13, and P14 liked the fact that each line comes with its own output as well as the Data Details, a clear difference from the traditional approach of writing a single expression pipeline and only seeing the final result: "What's cool is that what happens below [on the table] depends where you are on the [code line] so it's for each line. Usually I only look at beginning data and final output. The same for data details." (P5)

Participants generally made heavy use of clicking on lines to focus on particular transformation steps to validate their understanding of the code. While all participants made use of this core feature of Unravel, beginners clicked on lines more than experts. We found this to be true in the logs as well. Figure 6.9 visualizes the total lines clicked which shows how beginners clicked 341 times, and

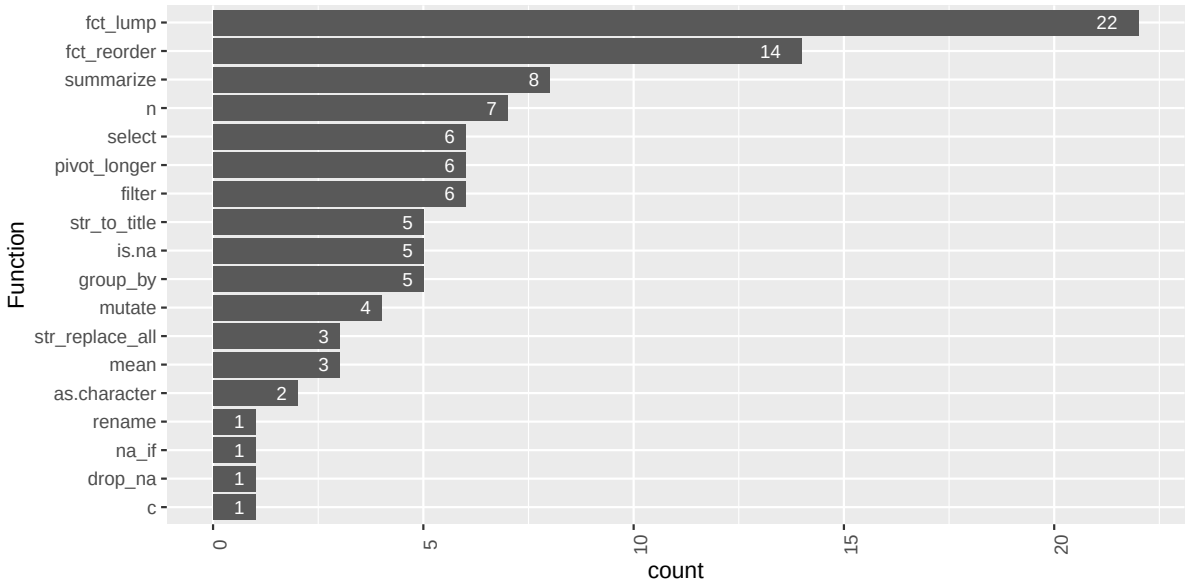

Figure 6.10 The count of Function Help clicks and the corresponding functions sorted by highest to lowest count.

experienced participants clicked 282 times. One trend with this result is that participants who clicked through the intermediate lines of data wrangling code generally completed more tasks besides two exceptions: P11 who was able to complete 2 tasks despite the low number of clicks, and P6 who was having difficulty learning tidyverse R style for the first time as a heavy base R user. Clicking on intermediate lines does not necessarily lead to more success as we observed some participants were confused despite going through steps of a data wrangling pipeline, and experienced participants can better "visualize data operations" (P12) requiring fewer inspections.

Understanding a function's purpose and behavior: Unravel was able to achieve D1 by providing several ways of understanding unfamiliar code.

The Function Help was rated as the most useful feature of Unravel (Table 6.2) and we observed most participants using it to understand functions. This feature helped both beginners and experts and according to the logs, beginners clicked on functions more (57) than experts (43) overall. Figure 6.10 shows the most commonly clicked functions which reveals an interesting pattern: some functions are more intuitive than others like `c`, `drop_na`, `na_if` and `rename`. While other functions are confusing like the `fct_lump` or `fct_reorder` functions which don't clearly convey its purpose, and `group_by` or `summarize` which participants found confusing since it's difficult to visualize aggregated computations and how `summarize` drastically changes shape was confusing. The `filter` was sometimes confused by some participants (P9, P7, P11, P10) to mean *filtering out* rows that meet a criteria when they found that it meant *selecting rows* that meet for the criteria. The `n` function in Task C was confusing to participants because it was unclear whether it was used to count the number of rows of a column or in each group's column.

Beginners like P1, P2, and P3 made a lot of inferences as to the purpose of functions since they

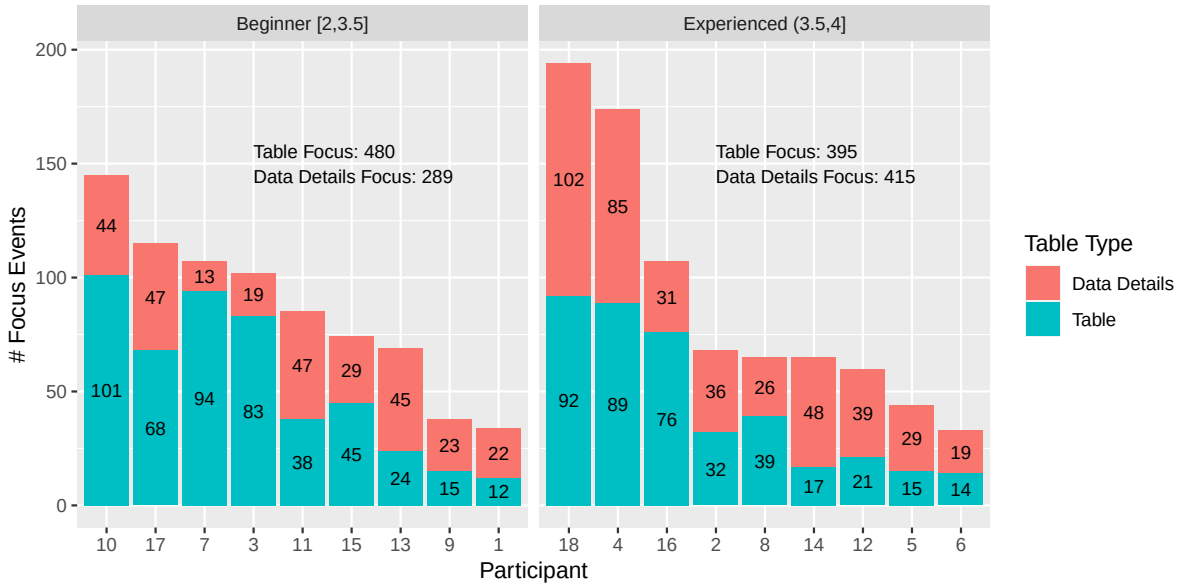

Figure 6.11 Number of times participants are focusing at the output Table and the Data Details.

were not too familiar with R and validated their assumption using the Function Help. For example, when clicking the `pivot_longer` on the Task C, P2 was happy to have refrenced the documentation: "Oh values_to! Very helpful to have a link to the documentation here." Experienced participants also made heavy use of the Function Help feature when dealing with unfamiliar functions specifically in the tidyverse R API. P6 admitted that they were "a base R person" and they clicked on several features around the `group_by` and `summarize` function which they were particular confused on for the Task B. Even for experienced participants, however, they still reached for function documentation for functions they don't use often. Other experts like P8 and P10 made use of the function click to help when they were confused by the `fct_lump` because they don't use it much in their coding, and `pivot_longer` "which is always confusing to visualize and memorize" (P10) despite having used it frequently in the past.

However, we also observed that validating one's understanding of code behavior required more than reading the documentation for functions. Beginners like P9 and P11 would often click on particular lines, and examine the interactive output of the dataframes at those lines to validate their assumptions about *how the code behaves*. They were able to make guesses based on how the output changed after making a change like reordering lines (P9), or viewing the function summary for the line (P11). P6 who was experienced in R but not in the tidyverse R discovered the behavior of functions when they tried to make a change that resulted in errors, for example, incorrectly using the `na_if` to replace values as NAs by not including it inside of a modifying function like `mutate`; in this process, P6 discovered the `mutate` function that helped them use `na_if` properly.

Understanding R code: Since participants had varying levels of experience with R, understanding code sometimes required other types of interactions with Unravel. While the Function Help feature was useful in situations where participants were confused about unfamiliar functions, it did

not help beginners such as P9 or P11 understand how data wrangling operations compose together in relation to the task of creating a visualization. In other words, the mere presence of the data wrangling code and being able to explore it interactively did not help these participants with lesser knowledge about how to work with data in general. In these cases, participants were able to guess and validate their assumptions by simply looking at the output. For example, both P1 and P2 who were quite inexperienced with R were able to guess how `filter` works based on the name and validating their assumption by flipping through the rows of the dataframe output table. Similarly, P3 was able to guess `as.integer` casts a vector to an integer by the name but confirmed this was the case by examining the type of the column after the operation. We will discuss some of the limitations of supporting novices in Section 6.7.

6.6.3 RQ2: How does Unravel help them identify data quality issues and debug data wrangling code?

Unravel helped participants identify data quality issues and code mistakes by using several features of the tool (D2).

Participants used both the Table and Data Details to validate assumptions. As shown in Figure 6.11, beginners made less use of the Data Details overall (300) compared to experienced participants (436). However, beginners made more use of the interactive Table (488) compared to experienced group (422). There were no major differences overall between beginners and experienced participants on how much they made use of both tables. However, we observed beginners exploring the tables heavily compared to experienced participants, who seemed to place greater importance on checking the data details. Several beginners (P1, P2, P7, P9, P11) used the output Table by clicking through the 'Next'/'Previous' buttons to flip through the rows of the dataframe when trying to validate assumptions for a particular column. For example, P11 flipped through some rows of the output table when they were investigating the -99 outlier in Task B and once that became laborious switched the Data Details which pointed out the value. P1 cleverly used the sorting feature on the output table to find the same outlier in Task B. Although hesitant at first, experienced participants like P4, P5, P8, P12 went directly to the Data Details for every task to save on time after having success with it initially, instead of doing manual checks.

Unravel helped check assumptions about potential issues through multiple views. We also noticed an interesting behavior across participants where they triangulated potential issues using multiple views. Participants examined descriptive statistics about the variables and the potential issues in the Data Details view, and went back to the table to look for values corresponding to those issues (e.g. by searching for specific values in the table). For example, P1 searched for the miscoded `NA` value of "-" in Task A for the `country` column in its expanded details within Data Details. Similarly, P2 made use of the search feature in Task C to validate whether the `crop` column contained values with an "_" in the name (`cocoa_beans`). P16 also made heavy use of the search feature to validate removing the "-" in the `country` column in Task A. When things still didn't make sense, they explored other lines, and their corresponding code, Table and Data Details. For example,

P14 saw how the unique counts for the `season` column in the last step was 43 instead of the original 45. Confused, they then checked the line before the `filter` step and saw the missing values in episode ratings from the missing bar which dropped 2 seasons in the step before. Through these various elements in Unravel, participants were able piece together and triangulate information from each of these views: "It's tough because the tool doesn't know what you are wanting to do, but at least you can check things to make sure you haven't totally broken your code" (P15)

At-a-glance visualizations provided quick validations of data characteristics. Participants made use of the many always-on visualizations within Unravel to help them validate transformations.

Participants used the Data Details view to get an overall sense of the columns and quickly catch issues. The unique number of elements and missingness in the Data Details overview table helped participants quickly identify and validate issues after transformations. P14 used the unique counts for the `season` column when trying to check how the counts dropped in Task B and was able to spot the missing values in episode ratings that dropped 2 seasons. Participants used the in-line histograms to spot extreme values. For example, P10, P12 and P13 looked at the histogram of the `total_cup_points` column in Task A and saw that it was heavily left skewed. They examined the expanded details for the column and noticed there was a 0 as a minimum value, and they were able to filter out that value. P14 was able to spot the incorrect ordering of the `group_by` after `summarize` by taking a peek at the histogram for the `season` column: "Yeah so that should definitely more than 14 and should be more of a uniform distribution."

Unravel helped debug data wrangling mistakes. Participants were able to debug data wrangling code using Unravel by using the Error visual cue for lines with errors. Often times, participants either added an extra transformation step in the data wrangling code or modified the existing lines that resulted in errors. To achieve D2, Unravel's design allows rendering the code even there are lines that cause errors, where lines with errors have the Error change schema, highlighting the summary box red and displaying the error message when hovering over it. Participants were able to benefit from this design because they sometimes made mistakes of their own. When doing Task A, P1 initially thought "I wonder if [the mutate lines] are out of order in terms of how you're supposed to do it in R", and they re-unraveled the code after switching their order. They saw the red boxes next to the lines and realized their mistake reading the error message: "that's probably not correct because I broke it. Ah, country not found." P4 made a mistake thinking the `country` column in Task A has to be grouped and they write a `group_by(country)` followed by a `summarize(total_cup_points = mean())`. After re-unraveling and seeing the broken line, they read the error message through the summary box and realize, "Ok it didn't like that because mean is missing an argument." Sometimes fixing code required a combination of breaking the data wrangling code and validating one's understanding of how to use the functions through Function Help feature. For example, P6 tried to use `na_if` function on its own line to filter out missing `episode_ratings` values for Task B, but upon re-unraveling the modified code saw the broken code: "Ok it doesn't like that. Why didn't it like that? Did I pipe it right?" They clicked the `na_if`

function on the Code Overlay and after reading the documentation realized "it's for vectors, not dataframes so we need the mutate function." Once they applied the fix they saw no errors and carried on.

6.7 Discussion and Future Work

We discuss the broad implications of our findings, future work on improving the interactive exploration of data wrangling code, and identify the ways in which Unravel could be adapted to various programming languages and contexts.

6.7.1 Better Support for Novice Data Scientists

We found that certain features of Unravel helped novice data scientists in our study successfully complete tasks, but the tool did not provide adequate support to complete all tasks. The Function Help was a key feature in helping them understand a function's purpose. Clicking on lines allowed participants to be able to focus on particular line's the code and output. However, as one participant put it, "[Unravel] was helpful in that I did not have to run every line to view the intermediate data, but I still have difficulty how to write data wrangling operations and visualizing them." (P9) While the scaffolded code in our tasks did help novices learn data wrangling in R by examining the example code through Unravel, there was a lack of support in helping them *write* data wrangling code, and be able to understand *why* certain operations are needed for the goal of producing a visualization. P12 expressed that "Unravel would be a great tool for teaching using PRIMM", a structured approach to planning programming lessons and activities using the Predict, Run, Investigate, Modify, and Make stages [Sen19]. We incorporated all of these stages besides Make, which requires writing brand new code. Zhi et al. [Zhi18] had a similar study setup and found that an educational programming game with "buggy code" is a promising teaching strategy, as demonstrated by lower completion times and solution code length in assessment puzzles. For our future work, we can look into adapting PRIMM and continuing the buggy code teaching strategy to better support novices.

6.7.2 Live Programming for Data Wrangling

As discussed in Section 6.6, we found that generally Unravel was able to fit into the workflow of data scientists in RStudio with some pain points. Participants in our study found it useful to be able to highlight data wrangling code and unravel it through the Addins and explore within the IDE. However, all participants desired more liveliness once they explored existing code and wanted to make modifications. For example, P10 said "if I could somehow edit code [in Unravel] that would be cool. Then you can see it updating in real time and you don't have to go back." P2 expressed how "having to highlight everything is a little bit weird. It would be really nice if you could it in the same vein as execute line on editor, it would be a lot more intuitive." These comments make sense given these limitations of Unravel. For the purpose of this study, we decided to leave out live programming because we were focused on understanding how Unravel and its existing features

can support data scientists during exploratory analysis, and due to the complexity of updating the state of the application when incorporating live updates. However, we can take inspiration of tools like Glinda [DeL21] which implements live programming using a domain-specific language for data science programming or DITL [Wan22] which also adds the idea of storing snapshots of successful data wrangling code to revisit for debugging purposes. Lerner [Ler20] also explored the idea of live projection boxes showing the current outputs within the editor itself which is another potential solution for a more fluid exploration. We could also take inspiration from Rill Developer [RD22], an exploratory data tool for SQL which uses similar ideas of automating data quality checks and displaying them live as a programmer write queries.

6.7.3 Exploring and Highlighting Data Quality Issues

Overall, participants found Data Details feature of Unravel to be valuable in identifying potential data quality issues but found it to be limiting in a couple of different ways.

Nudging users to validate data issues. One pain point that we observed throughout our study is that there is a lack of nudges to check the Data Details for steps in the data wrangling code. For example, P1 who was a beginner said that "if there were any problems I would like there to be some kind of highlight here like a nudge to click on Data Details". Experienced participants like P4 kept forgetting to check Data Details and similarly, P10 said "I think getting used to the fact that each operation has its own data details. Not sure why I kept forgetting, but that's very useful actually." In other words, while the Data Details was useful to identify potential issues with the data, there wasn't enough nudging on the user to check the view. For future work, we can improve the design of the Data Details such that we incorporate affordances like having a number on the tab, or highlighting a particular line with problems to catch a user's attention.

Providing flexible tools to audit problems. A related issue for highlighting problems with the data has to do with providing full and flexible details about each line in the data wrangling code. P5 mentioned how "One thing that would be super neat to include is what the points in the distribution correspond to" which is a limitation of the in-lined histograms for each column that others point out as well (P1, P3). For the in-line distribution, P5 and P10 mentioned how it would be convenient if one could also customize the type of plot to display for the in-line distribution such as a density plot. They also mentioned that showing duplicates of the data would be beneficial but warned that "it's not easy to handle because it could be duplicates of various columns not just whole row." Indeed, we had decided not to check for duplicates because of the fact that it depends on which columns. However, in our future work, we could provide additional options for both in-line plots and advanced checks such as duplicates according to certain columns.

Future work on data quality checks. Finally, based on our observations of participants' and their feedback, future work should study how to refine data quality checks and warnings for supporting data scientists in their data wrangling process. There are two usability issues with the way we designed the Data Details view that became clear during the study. Many participants (P1, P2, P3, P4, P5, P8, P18) spent time exploring columns that are irrelevant to the task yet attracted their attention

due to the missing values bar, or potential problems numbers. The always-on visualization and statistics for the columns of a dataframe could also become noisy given a large enough dataset. To mitigate, P1 desired "being able to dismiss warnings like dismissing warnings in the IDE. In Eclipse I can silence warnings I don't care about". Shome et al. [Sho22] explored "a novel catalogue of data smells that can be used to indicate early signs of problems or technical debt in machine learning systems." For future work, a catalogue of common data smells could be used to categorize the problems on the interface. However, assigning severity to issues and fixing them might always remain a human-in-the-loop effort. As the authors note "automation however comes at the cost of reduced transparency as minuscule changes to the input data can cause drastic changes in the trained model" and a similar sentiment is shared by the authors of `dataReporter` that automation of cleaning means "all power is given to the the computer with no human supervision, and investigators are less likely to make an active, case-specific choice regarding the handling of the potential errors".

6.7.4 Limitations

We limited the scope of Unravel, the extensions we added, and the user study in order to explore tools to help data scientists debug and fix data wrangling code.

6.7.4.0.1 Tool limitations:

First, we scoped Unravel to work with certain types of code. For example, Unravel can currently only unravel a variable that contains a dataframe, a one-liner function calls, and single-table data wrangling functions in a `dplyr` [Wic21] and `tidyr` [Wic19b] fluent code. The output of an operation in fluent code could produce lists and other structures worth unraveling. While it is possible and beneficial to render outputs like lists and plots, we currently only support dataframes as the primary outputs. Second, the set of data quality issues we chose to automatically check only represents a small portion of the types of quality issues that data scientists face at large. Future research should look into extending Shome et al. [Sho22]'s idea of paper smells from beyond machine learning.

6.7.4.0.2 Generalizability:

Our findings of the user study might not generalize to other communities. For example, our tool is focused on R and the tidyverse dialect. It is important to note that some aspects of our findings might be unique to this project and the R community. Future research should explore how a tool like Unravel can help the Python, Julia, or SQL communities.

6.7.4.0.3 Participation bias:

One potential limitation of our study is a self-selection bias in our interviewee sample. Our sample was selected to include those who were interested in learning more about data wrangling in R. Participants who already are involved with the #TidyTuesday project may have influenced our sample towards those with the strongest feelings about Unravel. We mitigated this issue by using

random sampling and recruiting people of different skill levels with data wrangling and R (e.g. P1–P3 were beginners in R).

6.7.4.0.4 Analysis methodology:

To derive themes from our user study, we used qualitative coding to analyze and interpret our data which is limited by theoretical sensitivity and the synthesis conducted by the researchers participating in that process. We followed the guidelines set by Carlson [Car10] and performed a single-event member check with our results. Our quantitative analysis is limited to events that we logged and are simply proxies for behavior and strategies used during the study. A more rigorous comparative study should be conducted in the future to understand how well Unravel helps data scientists to a control group for example.

6.8 Conclusion

We built extensions to Unravel, a tool that enables structured explorations of data wrangling code by adding Function Help, Data Details, and Code Overlay and Table enhancements. Data wrangling can be a tedious and error-prone process because it requires data scientists to meticulously explore the data for potential problems, apply numerous transformations, and validate the resulting changes. Through formative interviews, we identified several types of data quality issues and the manual checks that are required throughout the data wrangling. Through a user study with 18 data scientists, we found Unravel and the extensions we added helped data scientists better explore data wrangling code, triangulate and validate assumptions made about the code or data, and diagnose data wrangling code. We discuss the design implications and limitations of Unravel and future work for data wrangling tools for learners, educators, and programmers.

CHAPTER

7

CONCLUSION

The thesis statement of this dissertation is:

> Data wrangling is an important step in data science programming that requires data to be transformed into a form amenable for analysis. However, data wrangling is a time-consuming and error-prone process that requires a programmer to learn and correctly apply numerous data transformation techniques. Programmers can understand, explore, and debug data wrangling code flexibly when aided by just-in-time learning tools that accommodate multiple learning objectives.

I defended the claims of the thesis statement through four studies. In the first study (Chapter 3), I investigated the question of why is it difficult for programmers to learn another programming language? I conducted an empirical study of Stack Overflow questions across 18 different programming languages and semi-structured interviews with professional programmers [Shr20]. From our inspection of 450 Stack Overflow questions, there were 276 instances of interference that occurred due to faulty assumptions originating from knowledge about a different language. The interviews revealed that programmers make failed attempts to relate a new programming language with what they already know. Our findings inform design implications for technical authors, toolsmiths, and language designers, such as designing documentation and automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem.

In the second study (Chapter 4), I investigated how can we build an inclusive, welcoming online community of practice that unites data scientists in their collective efforts to become experts? I conducted a study on #TidyTuesday—a daily hashtag project for data scientists using R—as one

solution to this problem. I found that the participants were attracted to the rhythm provided by the project, the opportunity for professional development, and becoming part of the larger R community. Through #TidyTuesday, participants enhanced both technical and communication skills by learning from others, adopting best practices in R, and building an online presence. #TidyTuesday was effective in forming an online CoP by disseminating best practices, providing opportunities for curations to satisfy community needs, bootstrapping offline events and promoting an inclusive, welcoming community.

Following up on the second study, in Chapter 5 I studied how can we support data scientists interactively learn data wrangling programming, especially as they adapt code? Data scientists have adopted a popular design pattern in programming called the fluent interface for composing data wrangling code. The fluent interface works by combining multiple transformations on a data table—or dataframes—with a single chain of expressions, which produces an output. Although fluent code promotes legibility, the intermediate dataframes are lost, forcing data scientists to *unravel* the chain through tedious code edits and re-execution. I designed a tool called Unravel that enables structural edits via drag-and-drop and toggle switch interactions to help data scientists explore and understand fluent code. Data scientists can apply simple structural edits via drag-and-drop and toggle switch interactions to reorder and (un)comment lines. To help data scientists understand fluent code, Unravel provides function summaries and always-on visualizations highlighting important changes to a dataframe. We discuss the design motivations behind Unravel and how it helps understand and explore fluent code. In a first-use study with 26 data scientists, I found that Unravel facilitated diverse activities such as validating assumptions about the code or data, exploring alternatives, and revealing function behavior.

Finally, I then studied how we can use a just-in-time learning tool like Unravel to facilitate exploratory analysis work, especially when it comes to debugging data wrangling code and data (Chapter 6). Through formative interviews, we identified several types of data quality issues and the manual checks that are required throughout the data wrangling. I extended Unravel to incorporate a new view called Data Details, which displays descriptive statistics about the dataframe columns at each step of a transformation pipeline. and Code Overlay and Table enhancements. I also extended Unravel to support learning by adding Function Help, a feature that allows users to click on a function to open its documentation in the IDE. Through a user study with 18 data scientists, we found Unravel and the extensions we added helped data scientists better explore data wrangling code and data, triangulate and validate assumptions using multiple views, and diagnose data wrangling code by allowing them to pinpoint a broken transformation step. I discussed future work for data wrangling tools that support flexible, and interactive tools for learners, educators, and programmers.

7.1 Future Work

There are many avenues of research that are worth pursuing which are beyond the scope of this dissertation. Here are a few directions for related to just-in-time learning in data science:

- **Synthesizing data wrangling code.** This dissertation investigated the use of interactive tools to help data scientists explore existing code. But, data scientists are a diverse group of people, many of whom are not formally trained in programming. Future research can work on how to best synthesize data wrangling code that produces simple, readable and trustable code. How can we use program synthesis or programming by example approaches to help novice data scientists compose code?

- **Intelligent data smell tools.** This dissertation only began to scratch the surface of how we could detect data smells and reliably categorize them by type or severity. In Chapter 6, we touched on the challenges in this space because certain issues might not be automated away since it requires contextual understanding around the source of the data. However, there could be some promising research around categorizing them for data science programming in general. What if we had data smell linting in our IDEs? How would that change data science programming?

- **Learning data wrangling with AI pair programmers.** GitHub CoPilot has raised some interesting questions about the future of programming as we head towards more and more automation of code composition. One problem with data wrangling is that it can initially be very daunting to learn all of the patterns to shape data for analysis tools. Could a tool like CoPilot help in speeding up the learning process? Or, will it cause more harm to our understanding of code?

BIBLIOGRAPHY

[Ame19] Amershi, S. et al. “Software engineering for machine learning: A case study”. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 291–300.

[And06] Anderson, B. *Imagined communities: Reflections on the origin and spread of nationalism*. Verso books, 2006.

[Ant10] Antin, J. & Cheshire, C. “Readers Are Not Free-Riders: Reading as a Form of Participation on Wikipedia”. *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*. CSCW ’10. Savannah, Georgia, USA: Association for Computing Machinery, 2010, pp. 127–130.

[Arm07] Armstrong, D. J. & Hardgrave, B. C. “Understanding mindshift learning: the transition to object-oriented development”. *MIS Quarterly* (2007), pp. 453–474.

[Aue17] Aue, W. R. et al. “Evaluating mechanisms of proactive facilitation in cued recall”. *Journal of Memory and Language* **94** (2017), pp. 103–118.

[Aug16] Augspurger, T. *Modern Pandas (Part 5): Tidy Data*. 2016. URL: `https://tomaugspurger.github.io/modern-5-tidy.html`.

[Bac14] Bache, S. M. & Wickham, H. *magrittr: A Forward-Pipe Operator for R*. R package version 1.5. 2014.

[Bad05] Badre, D. & Wagner, A. D. “Frontal lobe mechanisms that resolve proactive interference”. *Cerebral Cortex* **15**.12 (2005), pp. 2003–2012.

[Bal18] Baltes, S. et al. “SOTorrent: studying the origin, evolution, and usage of Stack Overflow code snippets”. *CoRR* **abs/1809.02814** (2018). arXiv: `1809.02814`.

[Bar16] Barik, T. et al. “From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration”. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 211–221.

[Bar18a] Barik, T. et al. “How should compilers explain problems to developers?” *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. 2018, pp. 633–643.

[Bar18b] Bart, A. C. et al. “Reconciling the Promise and Pragmatics of Enhancing Computing Pedagogy with Data Science”. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 2018, pp. 1029–1034.

[Bar20] Baruffa, O. & Son, V. van. *Twitter for R programmers*. 2020. URL: `https://www.t4rstats.com/index.html`.

[Bel90] Bellamy, R. & Gilmore, D. “Programming plans: internal or external structures”. *Lines of Thinking: Reflections on the Psychology of Thought* **2** (1990), pp. 59–72.

[Ber17] Bernhardsson, E. *The eigenvector of "why we moved from language X to language Y"*. 2017. URL: https://erikbern.com/2017/03/15/the-eigenvector-of-why-we-moved-from-language-x-to-language-y.html.

[Ber14] Berry, Michael and Kölling, Michael. “The state of play: a notional machine for learning programming”. *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. 2014, pp. 21–26.

[Bie14] Bierman, G. et al. “Understanding TypeScript”. *ECOOP 2014 – Object-Oriented Programming*. Ed. by Jones, R. 2014, pp. 257–281.

[Bou81] Boulay, B. du et al. “The black box inside the glass box: presenting computing concepts to novices”. *International Journal of Man-Machine Studies* **14**.3 (1981), pp. 237–249.

[Bow11] Bower, M. & McIver, A. “Continual and explicit comparison to promote proactive facilitation during second computer language learning”. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’11. 2011, pp. 218–222.

[Bra09] Brandt, J. et al. “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code”. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2009, pp. 1589–1598.

[Bra19] Braun, V. et al. “Thematic Analysis”. *Handbook of Research Methods in Health Social Sciences*. Ed. by Liamputtong, P. Singapore: Springer Singapore, 2019, pp. 843–860.

[Bro83] Brooks, R. “Towards a theory of the comprehension of computer programs”. *International Journal of Man-Machine Studies* **18**.6 (1983), pp. 543–554.

[Bro00] Brown, J. *Yak Shaving*. 2000.

[Bry] Bryan, J. *gapminder: Data from Gapminder*. https://github.com/jennybc/gapminder, http://www.gapminder.org/data/, https://doi.org/10.5281/zenodo.594018.

[Bry05] Bryant, S. L. et al. “Becoming Wikipedian: Transformation of Participation in a Collaborative Online Encyclopedia”. *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*. GROUP ’05. Sanibel Island, Florida, USA: Association for Computing Machinery, 2005, pp. 1–10.

[Bur13] Burg, B. et al. “Interactive record/replay for web application debugging”. *Proceedings of the 26th Annual ACM Symposium on User Unterface Software and Technology*. 2013, pp. 473–484.

[Cam20] Campise, K. *Guide to Data Science Bootcamps — Complete listing of Bootcamps in the US*. 2020. URL: https://www.discoverdatascience.org/programs/data-science-bootcamps/.

[Car10] Carlson, J. A. “Avoiding traps in member checking.” *Qualitative Report* **15**.5 (2010), pp. 1102–1113.

[Bco] *Certified B Corporation*. URL: `https://bcorporation.net`.

[Cha21] Chang, W. et al. *shiny: Web Application Framework for R*. R package version 1.6.0.9000. 2021.

[Cha06] Charmaz, K. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[Cha20] Chattopadhyay, S. et al. “What’s Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities”. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–12.

[Pan] *Comparison with R / R libraries*. 2014. URL: `https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_r.html`.

[Cra11] Cranshaw, J. & Kittur, A. “The polymath project: lessons from a successful online collaboration in mathematics”. *Proceedings of the SIGCHI conference on human factors in computing systems*. 2011, pp. 1865–1874.

[Cre] *Create React App*. 2019. URL: `https://create-react-app.dev`.

[Cui12] Cui, A. et al. “Discover breaking events with popular hashtags in twitter”. *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2012, pp. 1794–1798.

[Dab12] Dabbish, L. et al. “Social coding in GitHub: transparency and collaboration in an open software repository”. *Proceedings of the ACM 2012 conference on computer supported cooperative work*. 2012, pp. 1277–1286.

[Dac96] Daconta, M. C. *Java for C/C++ Programmers*. Wiley New York, 1996.

[Dan12] Danielsiek, H. et al. “Detecting and understanding students’ misconceptions related to algorithms and data structures”. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM. 2012, pp. 21–26.

[Das03] Dasu, T. & Johnson, T. *Exploratory Data Mining and Data Cleaning*. Vol. 479. John Wiley & Sons, 2003.

[Dav12] Davenport, T. H. & Patil, D. “Data scientist”. *Harvard business review* **90**.5 (2012), pp. 70–76.

[DeL21] DeLine, R. A. “Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language”. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–11.

[Den08] Denny, P. et al. “Evaluating a new exam question: Parsons problems”. *Proceedings of the Fourth International Workshop on Computing Education Research*. 2008, pp. 113–124.

[Dét95] Détienne, F. “Design strategies and knowledge in object-oriented programming: effects of experience”. *Human–Computer Interaction* **10**.2-3 (1995), pp. 129–169.

[Dri] *Dribbble - Discover the World’s Top Designers & Creative Professionals*. URL: `https://dribbble.com`.

[Dro20] Drosos, I. et al. “Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists”. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–12.

[DB86] Du Boulay, B. “Some difficulties of learning to program”. *Journal of Educational Computing Research* **2**.1 (1986), pp. 57–73.

[Elb21] Elbers, B. *tidylog: Logging for ’dplyr’ and ’tidyr’ Functions*. R package version 1.0.2.9000. 2021.

[Eri08] Erickson, I. “The translucence of Twitter”. *Ethnographic praxis in industry conference proceedings*. Vol. 2008. 1. Wiley Online Library. 2008, pp. 64–78.

[Faa18] Faas, T. et al. “Watch Me Code: Programming Mentorship Communities on Twitch.Tv”. *Proc. ACM Hum.-Comput. Interact.* **2**.CSCW (2018).

[Fab22] Fabri, A. *pipediff: Show Diffs Between Piped Steps*. R package version 0.0.0.9000. 2022.

[Fer20] Ferdowsifard, K. et al. “Small-Step live programming by example”. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 614–626.

[Fie17] Fiesler, C. et al. “Growing their own: Legitimate peripheral participation for computational learning in an online fandom community”. *Proceedings of the 2017 ACM conference on computer supported cooperative work and social computing*. 2017, pp. 1375–1386.

[For16] Ford, D. et al. “Paradise Unplugged: Identifying Barriers for Female Participation on Stack Overflow”. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 846–857.

[For18] Ford, D. et al. ““We Don’t Do That Here”: How Collaborative Editing with Mentors Improves Engagement in Social Q&A Communities”. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: Association for Computing Machinery, 2018.

[Fow10] Fowler, M. *Domain-specific Languages*. Pearson Education, 2010.

[Fow05] Fowler, M. & Evans, E. “Fluent interface”. *martinfowler.com* (2005).

[Ger13] German, D. M. et al. “The evolution of the R software ecosystem”. *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE. 2013, pp. 243–252.

[Gil16] Gilbert, S. “Learning in a Twitter-based community of practice: an exploration of knowledge exchange as a motivation for participation in #hcsmca”. *Information, Communication & Society* **19**.9 (2016), pp. 1214–1232.

[Gil88] Gilmore, D. J. & Green, T. R. G. “Programming plans and programming expertise”. *The Quarterly Journal of Experimental Psychology Section A* **40**.3 (1988), pp. 423–442.

[Gla67] Glaser, B. G. & Strauss, A. L. “Grounded theory: Strategies for qualitative research”. *Chicago, lL: Aldine Publishing Company* (1967).

[Goe3] Goetz, M. *ways data preparation tools help you get ahead of big data. Forrester*. 3.

[Gra14] Graham, T. & Wright, S. “Discursive equality and everyday talk online: The impact of “superparticipants””. *Journal of Computer-Mediated Communication* **19**.3 (2014), pp. 625–642.

[Gre19] Greve, B. *A Beginner’s Guide to Clean Data: Practical advice to spot and avoid data quality problems*. 2019.

[Gru11] Gruzd, A. et al. “Imagining Twitter as an imagined community”. *American Behavioral Scientist* **55**.10 (2011), pp. 1294–1318.

[Guo13] Guo, P. J. “Online Python Tutor: embeddable web-based program visualization for CS education”. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM. 2013, pp. 579–584.

[Guo11] Guo, P. J. et al. “Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts”. *Proceedings of the 24th annual ACM symposium on User interface software and technology*. 2011, pp. 65–74.

[Guo12] Guo, P. J. “Software tools to facilitate research programming”. PhD thesis. Stanford University Stanford, CA, 2012.

[Har15] Hardin, J. et al. “Data science in statistics curricula: Preparing students to “think with data””. *The American Statistician* **69**.4 (2015), pp. 343–353.

[Hea19] Head, A. et al. “Managing messes in computational notebooks”. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–12.

[Hee21] Heer, J. *Data Wrangler*. 2021.

[Her20] Hertweck, K. *If you build it, they will come...but then what? Facilitating communities of practice in R*. 2020. URL: `https://rstudio.com/resources/rstudioconf-2020/if-you-build-it-they-will-come-but-then-what-facilitating-communities-of-practice-in-r/`.

[Hol19] Holikatti, M. et al. “Learning to Airbnb by engaging in online communities of practice”. *Proceedings of the ACM on Human-Computer Interaction* **3**.CSCW (2019), pp. 1–19.

[Hor20] Horst, A. M. et al. *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. R package version 0.1.0. 2020.

[Hou17] Hou, Y. & Wang, D. "Hacking with NPOs: collaborative analytics and broker roles in civic data hackathons". *Proceedings of the ACM on Human-Computer Interaction* **1**.CSCW (2017), pp. 1–16.

[Hou] Hould, J.-N. *Tidy Data in Python*. URL: `https://www.jeannicholashould.com/tidy-data-in-python.html`.

[Hub08] Huberman, B. A. et al. "Social networks that matter: Twitter under the microscope". *arXiv preprint arXiv:0812.1045* (2008).

[Hug] Hughes, E. & Ward, P. *TidyX*. URL: `https://www.youtube.com/channel/UCP8l94xtoemCH_GxByvTuFQ/`.

[Iha96] Ihaka, R. & Gentleman, R. "R: a language for data analysis and graphics". *Journal of computational and graphical statistics* **5**.3 (1996), pp. 299–314.

[Joh15] Johnson, B. et al. "Bespoke tools: adapted to the concepts developers know". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 878–881.

[Jon02] Jones, A. *C# for Java developers*. AOL Time Warner Book Group, 2002.

[Jon97] Jones, Q. "Virtual-communities, virtual settlements & cyber-archaeology: A theoretical outline". *Journal of Computer-Mediated Communication* **3**.3 (1997), JCMC331.

[Jon06] Jonides, J. & Nee, D. "Brain mechanisms of proactive interference in working memory". *Neuroscience* **139**.1 (2006), pp. 181–193.

[Jon98] Jonides, J. et al. "Inhibition in verbal working memory revealed by brain activation". *Proceedings of the National Academy of Sciences* **95**.14 (1998), pp. 8410–8413.

[Kac10] Kaczmarczyk, L. C. et al. "Identifying student misconceptions of programming". *Computer Science Education (SIGCSE)*. 2010, pp. 107–111.

[Kag17] Kaggle. *2017 Kaggle Machine learning & Data Science Survey*. 2017. URL: `https://www.kaggle.com/kaggle/kaggle-survey-2017`.

[Kag20] Kaggle. *State of Data Science and Machine learning 2020*. 2020. URL: `https://www.kaggle.com/kaggle-survey-2020`.

[Kan11] Kandel, S. et al. "Wrangler: Interactive visual specification of data transformation scripts". *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2011, pp. 3363–3372.

[Kan12] Kandel, S. et al. "Enterprise data analysis and visualization: An interview study". *IEEE Transactions on Visualization and Computer Graphics* **18**.12 (2012), pp. 2917–2926.

[Kel05] Kelleher, C. & Pausch, R. “Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers”. *ACM Computing Surveys (CSUR)* **37**.2 (2005), pp. 83–137.

[Ker17a] Kery, M. B. & Myers, B. A. “Exploring exploratory programming”. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2017, pp. 25–29.

[Ker17b] Kery, M. B. et al. “Variolite: Supporting exploratory programming by data scientists”. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017, pp. 1265–1276.

[Ker17c] Kery, M. B. et al. “Variolite: Supporting Exploratory Programming by Data Scientists”. *CHI*. Vol. 10. 2017, pp. 3025453–3025626.

[Ker18] Kery, M. B. et al. “The story in the notebook: Exploratory data science using a literate programming tool”. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–11.

[Ker19] Kery, M. B. et al. “Towards effective foraging by data scientists to find past analysis choices”. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–13.

[Ker20] Kery, M. B. et al. “mage: Fluid moves between code and graphical work in computational notebooks”. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 140–151.

[Kim16] Kim, M. et al. “The emerging role of data scientists on software development teams”. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 96–107.

[Ko04a] Ko, A. J. & Myers, B. A. “Designing the Whyline: A debugging interface for asking questions about program behavior”. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2004, pp. 151–158.

[Ko04b] Ko, A. J. et al. “Six learning barriers in end-user programming systems”. *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE. 2004, pp. 199–206.

[Ko11] Ko, A. J. et al. “The state of the art in end-user software engineering”. *ACM Computing Surveys (CSUR)* **43**.3 (2011), pp. 1–44.

[Kos14] Kostkova, P. et al. “#swineflu: The use of twitter as an early warning and risk communication tool in the 2009 swine flu pandemic”. *ACM Transactions on Management Information Systems (TMIS)* **5**.2 (2014), pp. 1–25.

[Kou17] Kou, Y. & Gray, C. M. “Supporting Distributed Critique through Interpretation and Sense-Making in an Online Creative Community”. *Proc. ACM Hum.-Comput. Interact.* **1**.CSCW (2017).

[Kou18] Kou, Y. et al. “Understanding Social Roles in an Online Community of Volatile Practice: A Study of User Experience Practitioners on Reddit”. *Trans. Soc. Comput.* **1**.4 (2018).

[Kri] Kriebel, A. & Eva, M. *Makeover Monday*. URL: https://www.makeovermonday.co.uk.

[Kro19] Kross, S. & Guo, P. J. "Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges". *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–14.

[Lan77] Landis, J. R. & Koch, G. G. "The measurement of observer agreement for categorical data". *Biometrics* (1977), pp. 159–174.

[Lan] *Langserver.org*. URL: https://langserver.org/.

[Lau21] Lau, S. et al. "TweakIt: Supporting end-user programmers who transmogrify code". *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–12.

[Lav91] Lave, J., Wenger, E., et al. *Situated learning: Legitimate peripheral participation*. Cambridge university press, 1991.

[Lee13] Lee, Y. Y. et al. "Drag-and-drop refactoring: Intuitive and efficient program transformation". *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 23–32.

[Ler20] Lerner, S. "Projection boxes: On-the-fly reconfigurable visualization for live programming". *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–7.

[Lie14] Lieber, T. et al. "Addressing misconceptions about code with always-on programming visualizations". *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2014, pp. 2481–2490.

[Lin79] Linger, R. C. et al. "Structured programming: theory and practice" (1979).

[Liu17] Liu, F. et al. "Selfies as social movements: Influences on participation and perceived impact on stereotypes". *Proceedings of the ACM on Human-Computer Interaction* **1**.CSCW (2017), p. 72.

[Loh17] Lohr, S. "Where the STEM jobs are (and where they aren't)". *New York Times* **1** (2017).

[Lon17] Long, W. *Analyzing GitHub, how developers change programming languages over time*. 2017. URL: https://blog.sourced.tech/post/language_migrations/.

[LK13] Loureiro-Koechlin, C. & Butcher, T. "The emergence of converging communities via Twitter". *The Journal of Community Informatics* **9**.3 (2013).

[Mar13a] Marlow, J. & Dabbish, L. "Activity traces and signals in software developer recruitment and hiring". *Proceedings of the 2013 conference on Computer supported cooperative work*. 2013, pp. 145–156.

[Mar14] Marlow, J. & Dabbish, L. “From Rookie to All-Star: Professional Development in a Graphic Design Social Networking Site”. *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’14. Baltimore, Maryland, USA: Association for Computing Machinery, 2014, pp. 922–933.

[Mar13b] Marlow, J. et al. “Impression formation in online peer production: activity traces and personal profiles in github”. *Proceedings of the 2013 conference on Computer supported cooperative work*. 2013, pp. 117–128.

[Mar18] Martin, K. G. *Preparing data for analysis is (more than) half the battle. Analysis Factor*. 2018.

[Mas10] Mason, M. “Sample size and saturation in PhD studies using qualitative interviews”. *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*. Vol. 11. 3. 2010.

[McM86] McMillan, D. W. & Chavis, D. M. “Sense of community: A definition and theory”. *Journal of community psychology* **14**.1 (1986), pp. 6–23.

[Mey13] Meyerovich, L. A. & Rabkin, A. S. “Empirical analysis of programming language adoption”. *ACM SIGPLAN Notices*. Vol. 48. 10. ACM. 2013, pp. 1–18.

[Mid20] Middleton, J. et al. “Data Analysts and Their Software Practices: A Profile of the Sabermetrics Community and Beyond”. *Proceedings of the ACM on Human-Computer Interaction* **4**.CSCW1 (2020), pp. 1–27.

[Moc18] Mock, T. *The Mockup Blog: TidyTuesday*. 2018.

[Moc22] Mock, T. *Tidy Tuesday: A weekly data project aimed at the R ecosystem*. 2022.

[Mos52] Moser, C. A. “Quota sampling”. *Journal of the Royal Statistical Society. Series A (General)* **115**.3 (1952), pp. 411–423.

[Mos] Mostipak, J. *Join the “R for Data Science” online learning community*. URL: `https://www.jessemaegan.com/post/join-the-r-for-data-science-online-learning-community/`.

[Mug14] Mugar, G. et al. “Planet Hunters and Seafloor Explorers: Legitimate Peripheral Participation through Practice Proxies in Online Citizen Science”. *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’14. Baltimore, Maryland, USA: Association for Computing Machinery, 2014, pp. 109–119.

[Mul19] Muller, M. et al. “How data science workers work with data: Discovery, capture, curation, design, creation”. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–15.

[Nel97] Nelson, H. J. et al. “Journeys up the mountain: different paths to learning object-oriented programming”. *Accounting, Management and Information Technologies* **7**.2 (1997), pp. 53–85.

[Nel09] Nelson, H. J. et al. “Patterns of transition: the shift from traditional to object-oriented development”. *Journal of Management Information Systems* **25**.4 (2009), pp. 271–298.

[Dat] *New Course: Python for R Users*. 2018. URL: `https://www.datacamp.com/community/blog/course-python-r-users`.

[Ni21] Ni, W. et al. “reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example”. *The 34th Annual ACM Symposium on User Interface Software and Technology*. 2021, pp. 258–269.

[Nie17] Niederer, C. et al. “TACO: Visualizing changes in tables over time”. *IEEE Transactions on Visualization and Computer Graphics* **24**.1 (2017), pp. 677–686.

[Nor00] Norvig, P. *Python for Lisp Programmers*. 2000. URL: `https://norvig.com/python-lisp.html`.

[Ohr17] Ohri, A. *Python for R Users: A Data Science Approach*. John Wiley & Sons, 2017.

[Par] Parr, T. *Clarifying exceptions and visualizing tensor operations in deep learning code*. URL: `https://explained.ai/tensor-sensor/index.html`.

[Pas19] Pasquini, L. A. & Eaton, P. W. “The# acadv Community: Networked Practices, Professional Development, and Ongoing Knowledge Sharing in Advising”. *NACADA Journal* **39**.1 (2019), pp. 101–115.

[Pas17] Passi, S. & Jackson, S. “Data Vision: Learning to See Through Algorithmic Abstraction”. *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. CSCW ’17. Portland, Oregon, USA: Association for Computing Machinery, 2017, pp. 2436–2447.

[Pas18] Passi, S. & Jackson, S. J. “Trust in Data Science: Collaboration, Translation, and Accountability in Corporate Data Science Projects”. *Proc. ACM Hum.-Comput. Interact.* **2**.CSCW (2018).

[Pen87] Pennington, N. “Stimulus structures and mental representations in expert comprehension of computer programs”. *Cognitive Psychology* **19**.3 (1987), pp. 295–341.

[Pip] *Pipe*. URL: `https://magrittr.tidyverse.org/reference/pipe.html`.

[Pos04a] Postle, B. R. & Brush, L. N. “The neural bases of the effects of item-nonspecific proactive interference in working memory”. *Cognitive, Affective, & Behavioral Neuroscience* **4**.3 (2004), pp. 379–392.

[Pos04b] Postle, B. R. et al. “Prefrontal cortex and the mediation of proactive interference in working memory”. *Cognitive, Affective, & Behavioral Neuroscience* **4**.4 (2004), pp. 600–608.

[Pot11] Potts, L. et al. “Tweeting disaster: hashtag constructions and collisions”. *Proceedings of the 29th ACM international conference on Design of communication*. 2011, pp. 235–240.

[Pu21] Pu, X. et al. “Datamations: Animated explanations of data analysis pipelines”. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–14.

[Mat] *Python for MATLAB Users*. URL: https://www.datacamp.com/courses/python-for-matlab-users.

[Qia17] Qian, Y. & Lehman, J. “Students’ misconceptions and other difficulties in introductory programming: a literature review”. *ACM Transactions on Computing Education (TOCE)* **18**.1 (2017), p. 1.

[Qua17] QuantEcon. *MATLAB–Python–Julia cheatsheet*. 2017. URL: https://cheatsheets.quantecon.org.

[Rla] *R-Ladies Global & R-Ladies is a world-wide organization to promote gender diversity in the R community*. URL: https://rladies.org.

[Rpr] *R: The R Project for Statistical Computing*. URL: https://www.r-project.org.

[Rat17] Rattenbury, T. et al. *Principles of data wrangling: Practical techniques for data preparation*. " O’Reilly Media, Inc.", 2017.

[Rus] *References and borrowings*. URL: https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html.

[Rey] Reynolds, G. *Tidy Tuesday Highlights*. URL: https://evamaerey.github.io/tidytuesday_walk_through/tidytuesday_highlights.html.

[Ric81] Rich, C. *Inspection Methods in Programming*. Tech. rep. TR-604. MIT, 1981.

[RD22] Rill Data, I. *Data Wrangler*. 2022.

[Ros20] Rosenberg, J. et al. “Becoming ‘Tidier’ Over Time: Studying# tidytuesday as a Social Media-Based Context for Learning to Visualize Data” (2020).

[Rsta] *RStudio - RStudio*. URL: https://rstudio.com/products/rstudio/.

[Rstb] *RStudio | Open source & professional software for data science teams - RStudio*. URL: https://rstudio.com.

[RSt20] RStudio, P. *RStudio Primers*. 2020. URL: https://rstudio.cloud/learn/primers.

[Rstc] *rstudio::conf*. 2020. URL: https://rstudio.com/resources/rstudioconf-2020/.

[Rul18] Rule, A. et al. “Exploration and explanation in computational notebooks”. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12.

[Blo] *Rust for C++ programmers*. 2014. URL: http://featherweightmusings.blogspot.com/2014/04/rust-for-c-programmers-part-1-hello.html.

[SL22] Sam Lau, P. G. *Pandas Tutor - visualize Python pandas code*. 2022.

[Sca08] Scaffidi, C. et al. “Topes”. *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE. 2008, pp. 1–10.

[Sch21] Schloerke, B. et al. *learnr: Interactive Tutorials for R*. https://rstudio.github.io/learnr/, https://github.com/rstudio/learnr. 2021.

[Sch90] Scholtz, J. & Wiedenbeck, S. “Learning second and subsequent programming languages: A problem of transfer”. *International Journal of Human–Computer Interaction* **2**.1 (1990), pp. 51–72. eprint: https://doi.org/10.1080/10447319009525970.

[SK22] Sean Kross, P. G. *Tidy Data Tutor - Visualize R tidyverse code data pipelines*. 2022.

[See17] Seering, J. et al. “Shaping pro and anti-social behavior on twitch through moderation and example-setting”. *Proceedings of the 2017 ACM conference on computer supported cooperative work and social computing*. 2017, pp. 111–125.

[Seg07] Segal, J. “Some problems of professional end user developers”. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE. 2007, pp. 111–118.

[Sen19] Sentance, S. et al. “Teaching computer programming with PRIMM: a sociocultural perspective”. *Computer Science Education* **29**.2-3 (2019), pp. 136–176.

[Shn79] Shneiderman, B. & Mayer, R. “Syntactic/semantic interactions in programmer behavior: a model and experimental results”. *Int’l J. Parallel Programming* **8**.3 (1979), pp. 219–238.

[Sho22] Shome, A. et al. “Data Smells in Public Datasets”. *arXiv preprint arXiv:2203.08007* (2022).

[Shr19] Shrestha, N. & Parnin, C. “Instrument designs for validating cross-language behavioral differences”. *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2019, pp. 205–209.

[Shr18] Shrestha, N. et al. “It’s like Python but: towards supporting transfer of programming language knowledge”. *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2018, pp. 177–185.

[Shr20] Shrestha, N. et al. “Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?” *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 691–701.

[Shr21a] Shrestha, N. et al. “Unravel: A Fluent Code Explorer for Data Wrangling”. *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology*. UIST ’21 (2021).

[Shr21b] Shrestha, N. et al. “Unravel: A Fluent Code Explorer for Data Wrangling”. *The 34th Annual ACM Symposium on User Interface Software and Technology*. 2021, pp. 198–207.

[Sla] *Slack*. URL: https://slack.com.

[Sol82] Soloway, E. et al. “Tapping into tacit programming knowledge”. *Proceedings of the 1982 Conference on Human Factors in Computing Systems*. CHI ’82. 1982, pp. 52–57.

[Ste15] Steinmacher, I. et al. “Social barriers faced by newcomers placing their first contribution in open source software projects”. *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*. 2015, pp. 1379–1392.

[Sut18] Sutton, C. et al. “Data diff: Interpretable, executable summaries of changes in distributions for data wrangling”. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 2279–2288.

[Swe03] Sweller, J. et al. “The expertise reversal effect” (2003).

[Swi18] Swidan, A. et al. “Programming misconceptions for school students”. *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM. 2018, pp. 151–159.

[Tau17] Tausczik, Y. & Wang, P. “To Share, or Not to Share? Community-Level Collaboration in Open Innovation Contests”. *Proceedings of the ACM on Human-Computer Interaction* **1**.CSCW (2017), pp. 1–23.

[Top] *The 50 Most Popular MOOCs of All Time*. 2020. URL: `https://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/`.

[Car] *The Carpentries*. URL: `https://carpentries.org`.

[Clo] *Thinking in Clojure for Java Programmers*. 2010. URL: `https://www.factual.com/blog/thinking-in-clojure-for-java-programmers-1/`.

[Tida] *Tidyverse*. URL: `https://www.tidyverse.org`.

[Tidb] *tidyweek*. 2018. URL: `https://GitHub.com/rfordatascience/tidyweek`.

[Rub] *To Ruby From Python*. URL: `https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-python/`.

[Ton07] Tongco, M. D. C. “Purposive sampling as a tool for informant selection”. *Ethnobotany Research and Applications* **5** (2007), pp. 147–158.

[Tre17] Treisman, R. *Yale to offer new major in data science*. 2017. URL: `https://yaledailynews.com/blog/2017/03/08/yale-to-offer-new-major-in-data-science/`.

[Ues19] Uesbeck, P. M. & Stefik, A. “A randomized controlled trial on the impact of polyglot programming in a database context”. *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*. Ed. by Barik, T. et al. Vol. 67. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 1:1–1:8.

[Und57] Underwood, B. J. “Interference and forgetting.” *Psychological Review* **64**.1 (1957), p. 49.

[Air] *Vacation Rentals, Homes, Experiences & Places - Airbnb*. URL: https://www.airbnb.com.

[Van18] Vanhooser, A. *UC Berkeley announces data science pipeline program for students*. 2018. URL: https://www.dailycal.org/2018/09/20/uc-berkeley-announces-data-science-pipeline-program-for-students/.

[Vas14a] Vasilescu, B. "Social aspects of collaboration in online software communities" (2014).

[Vas14b] Vasilescu, B. et al. "How Social Q&A Sites Are Changing Knowledge Sharing in Open Source Software Communities". *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW '14. Baltimore, Maryland, USA: Association for Computing Machinery, 2014, pp. 342–354.

[Wan19] Wang, A. Y. et al. "How data scientists use computational notebooks for real-time collaboration". *Proceedings of the ACM on Human-Computer Interaction* **3**.CSCW (2019), pp. 1–30.

[Wan22] Wang, A. Y. et al. "Diff in the Loop: Supporting Data Comparison in Exploratory Data Analysis". *CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–10.

[War22] Waring, E. et al. *skimr: Compact and Flexible Summaries of Data*. R package version 2.1.4. 2022.

[Was] Wastl, E. *Advent of Code*. URL: https://adventofcode.com.

[Way] Wayne, H. *Syntax highlighting is a waste of an information channel*. URL: https://buttondown.email/hillelwayne/archive/syntax-highlighting-is-a-waste-of-an-information/.

[Wei21] Weinman, N. et al. "Fork It: Supporting stateful alternatives in computational notebooks". *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–12.

[Wen99] Wenger, E. *Communities of practice: Learning, meaning, and identity*. Cambridge university press, 1999.

[Wen02a] Wenger, E. et al. *Cultivating communities of practice: A guide to managing knowledge*. Harvard Business Press, 2002.

[Wen02b] Wenger, E. et al. "Seven principles for cultivating communities of practice". *Cultivating Communities of Practice: a guide to managing knowledge* **4** (2002).

[Tab] *We're changing the way you think about data*. URL: https://www.tableau.com/trial/tableau-software.

[Wic16] Wickham, H. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[Wic17] Wickham, H. *R for Data Science*. O'Reilly, 2017.

[Wic19a] Wickham, H. *babynames: US Baby Names 1880-2017*. R package version 1.0.0. 2019.

[Wic19b] Wickham, H. & Henry, L. *tidyr: Tidy Messy Data*. R package version 1.0.0. 2019.

[Wic14] Wickham, H. et al. "Tidy data". *Journal of Statistical Software* **59**.10 (2014), pp. 1–23.

[Wic21] Wickham, H. et al. *dplyr: A Grammar of Data Manipulation*. R package version 1.0.4. 2021.

[Wil06] Wilson, G. "Software carpentry: getting scientists to write better code by making them more productive". *Computing in Science & Engineering* **8**.6 (2006), pp. 66–69.

[Wil18] Wilson, G. *The Tidynomicon: A Brief Introduction to R for Python Programmers*. 2018. URL: `https://gvwilson.github.io/tidynomicon/`.

[Wu90] Wu, Q. & Anderson, J. R. *Problem-solving transfer among programming languages*. Tech. rep. Carnegie Mellon University, 1990.

[Xu12] Xu, A. & Bailey, B. "What Do You Think? A Case Study of Benefit, Expectation, and Interaction in a Large Online Critique Community". *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW '12. Seattle, Washington, USA: Association for Computing Machinery, 2012, pp. 295–304.

[Xu14] Xu, A. et al. "Voyant: Generating Structured Feedback on Visual Designs Using a Crowd of Non-Experts". *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW '14. Baltimore, Maryland, USA: Association for Computing Machinery, 2014, pp. 1433–1444.

[Yan] Yan, Q. *Tidy Data with Python*. URL: `https://qiushi.rbind.io/post/python-tidy-data/`.

[Ye03] Ye, Y. & Kishida, K. "Toward an understanding of the motivation of open source software developers". *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE. 2003, pp. 419–429.

[Zag16] Zagalsky, A. et al. "How the R Community Creates and Curates Knowledge: A Comparative Study of Stack Overflow and Mailing Lists". *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 441–451.

[Zap11] Zappavigna, M. "Ambient affiliation: A linguistic perspective on Twitter". *New media & society* **13**.5 (2011), pp. 788–806.

[Zap12] Zappavigna, M. *Discourse of Twitter and social media: How we use language to create affiliation on the web*. Vol. 6. A&C Black, 2012.

[Zeg18] Zegura, E. et al. "Care and the practice of data science for social good". *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*. 2018, pp. 1–9.

[Zen19] Zeng, A. & Crichton, W. “Identifying barriers to adoption for Rust through online discourse”. *9th Workshop on Evaluation and Usability of Programming Languages and Tools*. 2019, p. 15.

[Zha20a] Zhang, A. X. et al. “How do data science workers collaborate? roles, workflows, and tools”. *Proceedings of the ACM on Human-Computer Interaction* **4**.CSCW1 (2020), pp. 1–23.

[Zha] Zhang, N. *Another Book on Data Science*. URL: `https://www.anotherbookondatascience.com`.

[Zha20b] Zhang, T. et al. “Interactive Program Synthesis by Augmented Examples”. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 627–648.

[Zhi18] Zhi, R. et al. “Exploring instructional support design in an educational game for K-12 computing education”. *Proceedings of the 49th ACM technical symposium on computer science education*. 2018, pp. 747–752.

[Zhu16] Zhu, H. et al. “A contingency view of transferring and adapting best practices within online communities”. *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. 2016, pp. 729–743.