TRACTUS: Understanding and Supporting Source Code Experimentation in Hypothesis-Driven Data Science

Krishna Subramanian **Johannes Maas** **Jan Borchers**

RWTH Aachen University
52074 Aachen, Germany
{krishna, borchers}@cs.rwth-aachen.de johannes.maas1@rwth-aachen.de

ABSTRACT

Data scientists experiment heavily with their code, compromising code quality to obtain insights faster. We observed ten data scientists perform hypothesis-driven data science tasks, and analyzed their coding, commenting, and analysis practice. We found that they have difficulty keeping track of their code experiments. When revisiting exploratory code to write production code later, they struggle to retrace their steps and capture the decisions made and insights obtained, and have to rerun code frequently. To address these issues, we designed TRACTUS, a system extending the popular RStudio IDE, that detects, tracks, and visualizes code experiments in hypothesis-driven data science tasks. TRACTUS helps recall decisions and insights by grouping code experiments into *hypotheses*, and structuring information like code execution output and documentation. Our user studies show how TRACTUS improves data scientists' workflows, and suggest additional opportunities for improvement. TRACTUS is available as an open source RStudio IDE addin at http://hci.rwth-aachen.de/tractus.

Author Keywords

Data Science; Programming IDE; Exploratory programming; Information visualization; Observational study.

CCS Concepts

•**Human-centered computing** → **Information visualization;** *Web-based interaction; User interface design;* User studies;

INTRODUCTION

Every day, millions of data scientists use textual programming to obtain insights from data [25]. In their work, they follow an *exploratory programming* practice, which involves experimentation through source code to test ideas [20]. Since such experimentation leads to messy code, data scientists often rewrite their code to make it reusable, i.e., write production code [20, 28]. Additionally, data scientists document their code as well as *insights* obtained during their work and *rationale* that justifies their analysis methods [1, 29].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHI '20, April 25–30, 2020, Honolulu, HI, USA.
2020 Association of Computing Machinery.
ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376764

Writing production code by retracing code experiments, and then documenting insights and rationale can be laborious, as experimental code is hard to understand and navigate. To understand how this is currently done, we observed ten academic data scientists perform data science tasks. We qualitatively coded the observations to understand our participants' workflow. Many participants do not capture insights and rationale during experimentation, but recall these later by frequently *re-executing* source code. We also identified how participants organize their source code, explore ideas through code, and what information they use to rationalize their approach when writing production code later.

To address the problems we identified, we propose an algorithmic and visualization solution that builds on our findings. This solution (a) identifies and tracks the data scientist's code experiments, (b) groups these experiments into meaningful units, *hypotheses,* and (c) captures information that can help data scientists report insights and rationale. It visualizes this information alongside code, allowing users to interact with it. To realize our idea, we built TRACTUS, an addin for RStudio[1], a prominent statistical programming IDE [36].

This paper thus makes the following contributions:

- results of a video analysis that help understand data scientists' *workflow* during exploration and when rewriting code;
- TRACTUS, an RStudio addin that identifies, tracks, and visualizes code experiments grouped by hypotheses, and contextual information that can help data scientists recall rationale and insights; and
- results of two validations of TRACTUS that show how it can improve data science workflow.

BACKGROUND AND RELATED WORK

In this section, we review background information and prior research on data science, exploratory programming practice, source code visualization, and history navigation.

Analysis Paradigms and Transparent Statistics

There are two paradigms of hypothesis-driven data analysis: (a) exploratory data analysis (EDA) and (b) confirmatory analysis [33]. EDA is used in several research fields to generate insights from data [22]. In HCI, EDA is often used in exploratory studies [7] and domains like data visualization [11].

[1] http://rstudio.com

Confirmatory analysis complements EDA, and is usually used after EDA [33]. It involves using methods like null hypothesis significance testing (NHST), estimation using 95% confidence intervals, and regression analysis.

NHST is one of the most prevalent methods for validating research hypotheses [5, 7]. It involves computing *p*-values and using them as thresholds to validate hypotheses [23]. It is often employed in dichotomous testing, where the researcher would accept or reject a hypothesis on the basis of statistical significance [9]. Over the past decade, NHST has garnered a lot of criticism in HCI [7, 9]. One critique is *HARKing,* i.e., Hypothesizing After the Results are Known [7, 17], also known as "*p*-hacking" [12], "fishing" [16], or "wandering down the garden of forking paths" [12]. It refers to a situation where the researcher tries many analyses, but reports only the final, *successful* analysis. Like Gelman [12] and Pu et al. [27], we believe that HARKing is unintentional, and that it is a design problem. Omitting parts of the analyses from reports also leads to a lack of *transparency* [10, 30].

Data Science Programming and Tool Support

Data science is a broad term that refers to tasks in which information and knowledge are extracted from data [26]. During such tasks, there is usually no clear end goal; the data scientist experiments with her ideas to identify goals [20]. In hypothesis-driven data science, these ideas are the *hypotheses* the data scientist comes up with. Prior research has identified that during experimentation, data scientists practice code cloning and use informal versioning like comments to maintain code alternatives [18]. Since experimentation often leads to messy code, data scientists need to rewrite code to make it maintainable and reusable [20, 28]. Since cleaning up these messes can be hard, analysts often use the execution output to identify and understand code [14].

Prior research has produced several artifacts to help data scientists. Burrito [13] captures and displays source code outputs, timeline of activities, and notes from a data science project to help data scientists capture their data science workflow. Variolite [18] is a lightweight version-controlling system that helps data scientists maintain code alternatives and track outputs. More recently, Code Gathering Tools (CGT) [14] is an interactive extension to Python notebooks that can help data scientists find, clean, and manage code. Verdant [19] is also a notebook plugin that visualizes code history to help programmers find prior code. Unlike these tools, TRACTUS tracks the experiments by *grouping* them into *hypotheses,* and presents this structured visualization to help the user stay oriented.

Visualizations of Source Code and History

Source code visualization is a well explored area of research. Systems like Code Bubbles [4], Code Thumbnails [8], and Stacksplorer [21] visualize code to help improve comprehension and navigation. Programming IDEs employ other forms of visualization like icons and graphical overlays next to the code to encode information like syntax highlighting, code conventions, and version control information [32]. In data science, an important task is tracking the sources of data, i.e., data provenance. Provenance Explorer is a tool that supports

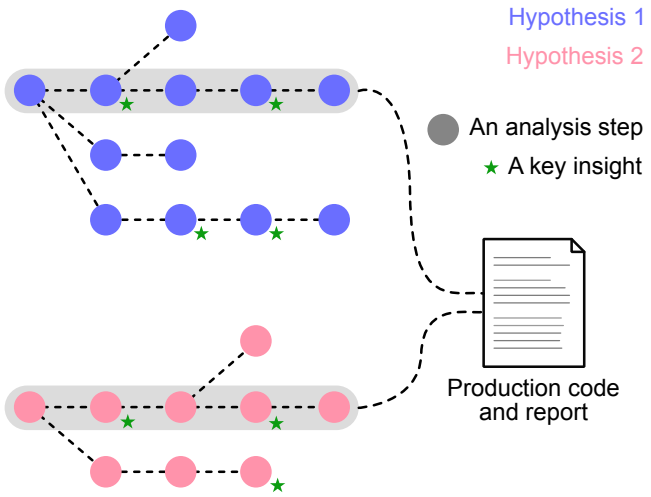

Figure 1. During data analysis, data scientists validate several hypotheses. This involves several steps like loading data, viewing descriptive statistics, and confirmatory analysis. Analysis generates insights, which may lead to further analyses. After experimentation, data scientists write production code and/or reports.

this task by visualizing the data and events associated with it as a graph [6]. Prior research artifacts, e.g., [19, 38, 39], visualize source code *history* to improve code comprehension and foraging.

Terminology

To support our discussion, we define the following terms:

• *Hypothesis:* A concrete, binary statement that the data scientist aims to validate. Hypotheses can be seen as the building blocks of analysis.

• *Step:* A high-level, meaningful task in data science, e.g., loading a dataset, viewing data characteristics, and testing the effect.

• *Analysis:* A collective term for the various steps that constitute validating a hypothesis.

• *Alternative Step and Alternative Analysis:* A variant of an analysis step and analysis. The selection of an alternative step often leads to an alternative analysis. E.g., removing an outlier might require using a non-parametric test.

• *Rationale:* The data scientist's justification for the methods chosen during the analysis.

• *Insight:* Information or knowledge obtained from data that the data scientist wants to disseminate.

• *Exploratory and Confirmatory Phases:* The exploratory phase is the initial analysis phase that follows exploratory programming practice. The confirmatory phase (not confirmatory *analysis* [33]) involves writing production code and reports.

MOTIVATIONAL STUDY

Data Collection

We collected observational videos from ten academic data scientists. All participants reported to have prior experience

(median = 2 years, range = 0.5 to 10 years) using RStudio for data science. We aimed to improve the external validity of our data by collecting videos from participants of varied experience and backgrounds like Numerical Analysis, Applied Psychology, and HCI. In the ensuing discussion, we will refer to our participants as P01–P10. Seven videos were recorded in our lab and three at the participant's workplace. Six participants analyzed fabricated data comparable to a real-world task (details in supplements), while others used data from their work. During the recording, participants were encouraged to think aloud. After the session, the experimenter clarified any questions that came up during the observation. We logged the video and audio of the session. We collected approximately 8 hours of content (median = 54 min.).

Method

The first author watched the videos to extract clips that met one or more of the following criteria: (a) participant interacts with RStudio, (b) participant interacts with another app to conduct analysis, e.g., does a web search on analysis procedure, and (c) participant thinks aloud about analysis. After performing an initial analysis on these clips, we generated three tiers of *process codes* [3]. These codes were used to categorize (1) domain-agnostic *programming tasks*, e.g., writing comments, creating a new file, or cloning code; (2) tasks in *analysis*, e.g., computing descriptive statistics, visualizing data, or building models; and (3) steps in *exploratory workflow*, e.g., creating alternatives, writing production code, or searching for code. Since our goal was to better understand data scientists' workflow, not provide a statistical breakdown of it, we used a qualitative analysis methodology.

FINDINGS

In this section, we describe our participants' workflow from exploratory to confirmatory phase based on our video analysis. Since our participants were from academia, we recommend to refrain from generalizing our findings to all data scientists.

How do data scientists experiment through code?

Many participants (P01, P03, and P07–10) used consoles to begin their analysis and then eventually documented source code in scripts.

All participants used a standard routine (Fig. 2) to explore alternatives: (1) *Find* and *clone* base code, (2) *contextualize* code, and (3) *evaluate* state. Base code, such as code from previous or current analyses or samples from the web, was thus crucial to kick-start code experiments. As one can expect, such code experiments were not conducted in a reusable, modular fashion. Also, most participants (P01, P04, and P06–10) did not report using any functions or modular code in their work. This is a known finding from prior research [18, 31].

After cloning, participants used the names and values of variables in the current session to update the arguments in the clone to suit their new exploration. As the final step in this routine, participants executed the source code to evaluate its state. This led to comparison of alternative explorations, insights, and helped determine next steps.

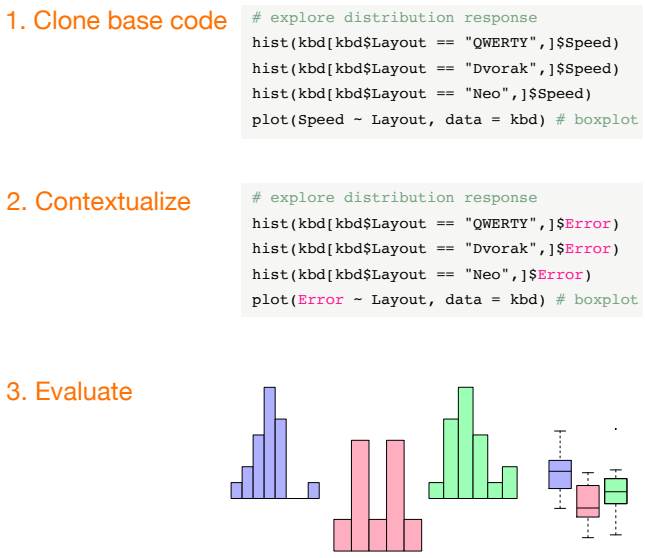

Figure 2. Data scientists follow an exploration *routine:* Clone base code, view the context of current dataset to modify the arguments of cloned code, and execute code to determine its state.

How is the source code organized?

Most participants (P02–04 and P06–10) organized their source code into *blocks*. Each block represented one meaningful step in the data science task, e.g., loading data or cleaning up data. Blocks were usually prefixed with a small descriptive comment, usually a high-level description of the task. P01 and P05 used code blocks infrequently, and reported that they do not always use it. While documentary structures like white spaces and comments are used to improve code readability [35], some participants (P01, P04, and P08) reported using blocks to be able to navigate source code later:

> *"[These] blobs [i.e., blocks] are useful when I go through [i.e., review] the source code. [They] help me parse code easier."* –P04

How are hypotheses validated and what leads to alternative analyses?

As stated before, a hypothesis is a binary statement that expresses the relationship between two or more variables. All participants performed significance tests during our observation. In addition to the significance test, other steps that constitute hypothesis validation are visualizing data, computing descriptive statistics, performing tests for statistical assumptions, and performing post-hoc tests. For significance tests, hypotheses were primarily expressed using R's formula notation[2]. The simplest notation is of the form, `measure ~ factor`, which refers to the hypothesis that investigates whether the factor has a significant effect on the measurement. While code corresponding to the other steps also used this

[2] http://tinyurl.com/y5of72lr

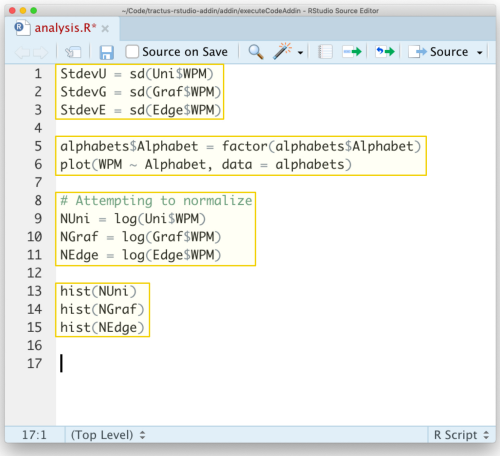

Figure 3. Reconstruction of the analysis code written by P08. Data scientists use code blocks, sometimes prefixed with a descriptive comment, to group meaningful steps in the analysis.

notation, column selection and dataset manipulation operations were more prevalent. E.g., P01 performed a test for normality using, `shapiro.test(data[data$method == "Unistrokes",].speed)`, where `Unistrokes` is a level of the factor, `method`, and `speed` is the measurement. (The participant analyzed a dataset that compared text entry techniques in mobile phones.) The statement would therefore be part of validating the hypothesis, `WPM ~ method`, i.e., investigating whether typing methods have an effect on the typing speed.

In addition to validating several hypotheses, participants also performed multiple analyses to validate the same hypothesis. We found that several participants (P01, P02, P04, and P10) conducted these alternative analyses after the data was modified, e.g., by transformations or outlier removals. This is not surprising, since the analysis method is almost entirely dependent on the data characteristics [12]. These changes to data mostly resulted from obtaining insights in the analysis, e.g., learning that data is log-normally distributed or that a certain test would not be valid for the situation. However, there were a few instances where the data was modified impulsively by participants:

> *"I will just [see] what happens to distribution when these [data] points are removed."* –P02

Participants used variable names like `logData` and `data_new` to track the different versions of data.

How do data scientists rationalize their analysis?

As discussed in the previous section, data scientists report the rationale for the decisions made in their work, along with the key insights. Participants used the following information to rationalize analysis decisions:

- Most participants (P01, P05–10) had *predetermined* one or more analysis steps, often based on their prior experience. E.g., before performing analysis, P07 knew that one of the factors in his data had three levels:

  > *"I will probably be doing an ANOVA test [sic] here, followed by pairwise comparisons."* –P07

  Participants do not capture this information *explicitly* during exploration, but later include them through documentation when writing production code.

- All participants used previous execution *results* as rationale, since results often lead to new insights about data. E.g., P08 used a quantile-quantile plot to rationalize the use of a non-parametric test.

- Some participants (P02, P03, P06, and P09) used resources, e.g., web articles[3], as rationale. These were later documented in the production code using comments.

How do data scientists track data insights?

While insights result from executing source code, it is often more than then results themselves, and includes the analyst's interpretation. Thus, insights were often detailed and too verbose to be captured as comments. E.g., an insight generated by P08 is:

> *"I would recommend [users to] use EdgeWrite [a text-input technique] here because the variance [of typing speed] is low, but one can also use Graffiti which has a higher average."* –P08

Only P04 and P05 used RMarkdown notebooks to track such verbose insights. P06 and P09 used comments with abstract information to document insights (e.g., `Test is inconclusive, see model o/p`); this abstraction leads to information loss. However, most participants (P01–03, P07, P08, and P10) did not use comments to document insights. They relied upon their short-term memory instead:

> *"The information [about insights from exploration] is something I still have in my head and it's usually [just] a few key insights."* –P07

Except for P04 and P05 who used RMarkdown notebooks, all participants had to *re-execute* current code, often several times, when writing production code to recall rationale and insights. This shows that data scientists overlook the need for capturing information on regular intervals during exploration.

Even during exploration, some participants found it difficult to keep track of the source code that produced a data insight, which often leads to re-executions. E.g., P07, who could not find a code snippet she was looking for during exploration, uttered:

> *"One of these three distributions is not normal... where is the line [of code] where [sic] I computed [i.e., plotted] the histograms?"* –P07

[3] E.g., https://stats.idre.ucla.edu/other/mult-pkg/whatstat/

What do data scientists use comments for?

Comments were used for documenting insights and rationale; navigation; and managing alternatives. While all participants used comments for these purposes in production code, some participants (P01, P07, P08, and P10) were reluctant to use comments during exploration:

> *"I write comments [only] when I have found something interesting [i.e., an insight]."* –P08

This is an implication of the exploratory programming practice, in which the focus is on getting results faster. Comments in production code were used to provide a high-level task description, e.g., `apply ANOVA` (P06). Some participants P02–04 used comments to describe what was programmatically done, e.g., `loop through each data segment...` (P02). Several participants (P02, P03, P05, P07, and P08) used comments to also capture rationale and insights in production code, e.g., `Preconditions for wilcox test are met` (P07).

P02 used stylized comments to distinguish comments about insights from other comments. P03 used section comments[4] for task descriptions, to navigate code more easily. There were some individual differences in frequency and style, e.g., length, verbosity, and use of inline vs. tail comments. Some participants (P3, P4, and P7) used comments to temporarily disable code snippets.

How do data scientists rewrite source code?

After exploring alternatives to obtain data insights, participants rewrite exploratory code to be able to reuse it. This code will be disseminated and/or stored for later. Participants rewrote code in two ways: (1) Clean up current code (P01, P02, P04–06, and P09) and (2) Rewrite code from scratch (P03, P07, P08, and P10). To prune current code, participants used code blocks, comments, and variable names to understand which source code to keep. Additionally, to identify relevant code snippets, participants re-executed source code, a behavior also exhibited during exploration. Participants often changed variable names when temporary names were used, added or modified comments, and rearranged code. Rewriting code required participants to retrace their steps by viewing the current exploratory code, prior executions in the console, and the history of commands. The relevant code is identified via results shown in the console and comments, if used. This code is then often cloned into a new file, and arguments are modified when appropriate.

Participants often found it difficult (1) to find the correct version of the source code and (2) to make sure that the execution dependencies of the code were intact. E.g., after validating several hypotheses, P07 wanted to move the code used to validate a hypothesis to a new file. He looked through his code to find relevant code, but upon pasting it into the new file and executing it, he found that an earlier statement that was used to set one column variable as a factor was not copied. This led to faulty execution.

[4] https://support.rstudio.com/hc/en-us/articles/200484568-Code-Folding-and-Sections

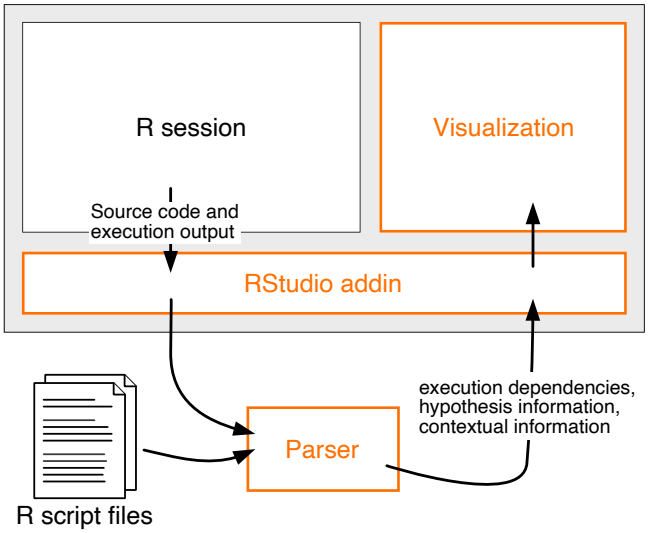

Figure 4. TRACTUS consists of three components: RStudio addin, parser, and the visualization. The *RStudio addin* feeds the R code and execution output from the R session to the parser. The *parser* breaks down the code, detects the hypothesis that the code belongs to, and finds execution dependencies in code. This information is then visualized.

Summary of Findings

1. During exploration and while rewriting code, data scientists have difficulty keeping track of the code that produced data insights and the states of code experiments.
2. Exploration involves a standard routine of *finding* base code, *cloning, contextualizing,* and *evaluating* it.
3. Hypotheses are the building blocks of analysis. Source code written to validate hypotheses have syntactic signals that make them detectable. Data manipulations lead to alternative analyses, and data scientists have to remember variable names to keep track of data versions.
4. Data scientists organize their code into blocks when writing code; these are used as checkpoints for navigation later.
5. Data scientists use (a) prior knowledge of statistical procedure, (b) text & graphic output of source code, and (c) external resources like webpages to rationalize their analysis.
6. Data scientists do not capture data insights initially, but instead rely on their memory and sparse documentation.
7. It is hard for data scientists to track the data dependencies in their code. This leads to faulty executions in production code.
8. Data scientists *rerun* code frequently to *recall* rationale, insights, and the states of explorations.

TRACTUS

To mitigate the problems we identified in our formative study, we present an interactive application, TRACTUS, that can help data scientists track source code that yielded insights during exploration (Finding #1) and understand their source code

explorations better when writing production code and reports later (Finding #8). The resulting system can reduce code re-runs, as well as help data scientists manage explorations, rewrite code for reuse, and write reports. We first provide an overview of TRACTUS, discuss the details of implementation and interaction design, and then describe TRACTUS' architecture.

Tractus consists of three components as shown in Fig. 5:

1. The **parser** is the back-end of TRACTUS, which breaks down R source code to obtain (a) the *hypotheses* investigated by the data scientist during analysis, (b) the execution *dependencies* among variables in source code, and (c) *contextual information* in source code such as the block and tail comments. In addition to comments, execution output and the order of execution are sent to the parser by our RStudio addin. Hypotheses are the atomic building blocks of analysis (Finding #3) and mimic the data scientist's thought process. Execution dependencies are captured to help minimize incorrect and faulty production code execution (Finding #7). Contextual information helps data scientists rationalize their analysis (Finding #5) and rewrite source code after exploration (Finding #6).
2. The **web app** or **visualization,** which acts as the front-end of TRACTUS as shown in Fig. 5. The web app receives information about source code groupings according to the hypothesis that is tested, execution dependencies, and contextual information from the parser, and *visualizes* it in real-time. It monitors the parser output for changes and updates the visualization when necessary. In the visualization, the source code is organized into blocks to improve navigation (Finding #4), and variables used in the analysis are emphasized to help track data provenance (Finding #3). Furthermore, based on our Finding #2, the visualization also supports *data injection.* This allows data scientists to select a block of code and modify the dependent and independent variables in it. The visualization can be shown in the RStudio viewer pane or in a web browser. Since RStudio's viewer pane does not support certain features like autocomplete or copy-to-clipboard, web browsers might be preferable.
3. The **RStudio addin**, which integrates the parser and web app into the R session. The addin watches the R session for new source code executions, captures them, and feeds them to the parser along with execution results. Note that only the valid statements, i.e., statements that successfully execute, are sent to the parser. The addin is also responsible for displaying the web app (i.e., the visualization front-end of TRACTUS) in the viewer pane of RStudio.

We designed TRACTUS in an iterative manner, gathering feedback from R analysts at every stage. After low-fidelity sketches to evaluate the visualization, we built two high-fidelity implementations (Fig. 6 and Fig. 5) to evaluate both the interaction and the visualization. We will now describe the parser and visualization in detail.

Parser

The parser is the back-end of TRACTUS that is responsible for detecting key information from the R source code. The parser is agnostic to the source of the R code—it could be an R script file, an R session's history database file, or raw source code fed in via the RStudio addin. Tracking the R session's history allows TRACTUS to capture code experiments that are done via console, a common practice among our participants. We validated the parser by using it to parse existing R scripts; we discuss the validation results at the end of this subsection.

Detecting Components of a Statement

The parser deconstructs the given R source code into an Abstract Syntax Tree (AST) [2] representation. The AST reveals the components of each statement such as the variable, expression, function name, and arguments (name and value). Then, the parser filters out statements that do not have to be visualized like package installations, statements that do not execute successfully, and control structures like loops and conditions. Unlike existing parsers, our custom parser captures comments (both inline and block) and line feeds in order to later detect code blocks.

Detecting Execution Dependencies

Detection of execution dependencies is not new [15, 37]. Execution dependencies are detected by tracking variables and statements. A statement that uses a variable depends on the statement that defined or modified that variable. Statements that use multiple variables depend on multiple statements; conversely, a variable can be depended upon by multiple statements. The parser ignores dependencies in control structures, e.g., dependencies from statements that are inside an if block to those outside the if block. Our parser validation revealed few instances of this, since hypothesis testing typically has a linear, albeit branching, control flow.

In addition to helping users understand their explorations better, revealing the execution dependencies also helps capture the alternative explorations that result from data modifications (Finding #3). Alternate explorations use a different data and are tracked in our visualization more easily (Fig. 5g).

The parser first keeps track of the variables resulting from the AST representation and then uses this information to cumulatively detect execution dependencies in the code. These dependencies are captured by the parser as a labeled Directed Acyclic Graph (DAG), in which each node is a statement and each directed edge is labeled with the variable name that establishes the dependency between the connected statements. A simple traversal of this graph results in all statements required to execute a statement with correct values.

Detecting Hypotheses

Data analysis using hypothesis testing usually consists of several hypotheses, each with possible alternative explorations (Finding #3). The parser detects hypotheses in statements by exploiting R's formula notation, data selection, and data manipulation operations (Finding #3).

In R, there are certain significance tests that allow users to specify hypothesis without this special formula notation, e.g.,

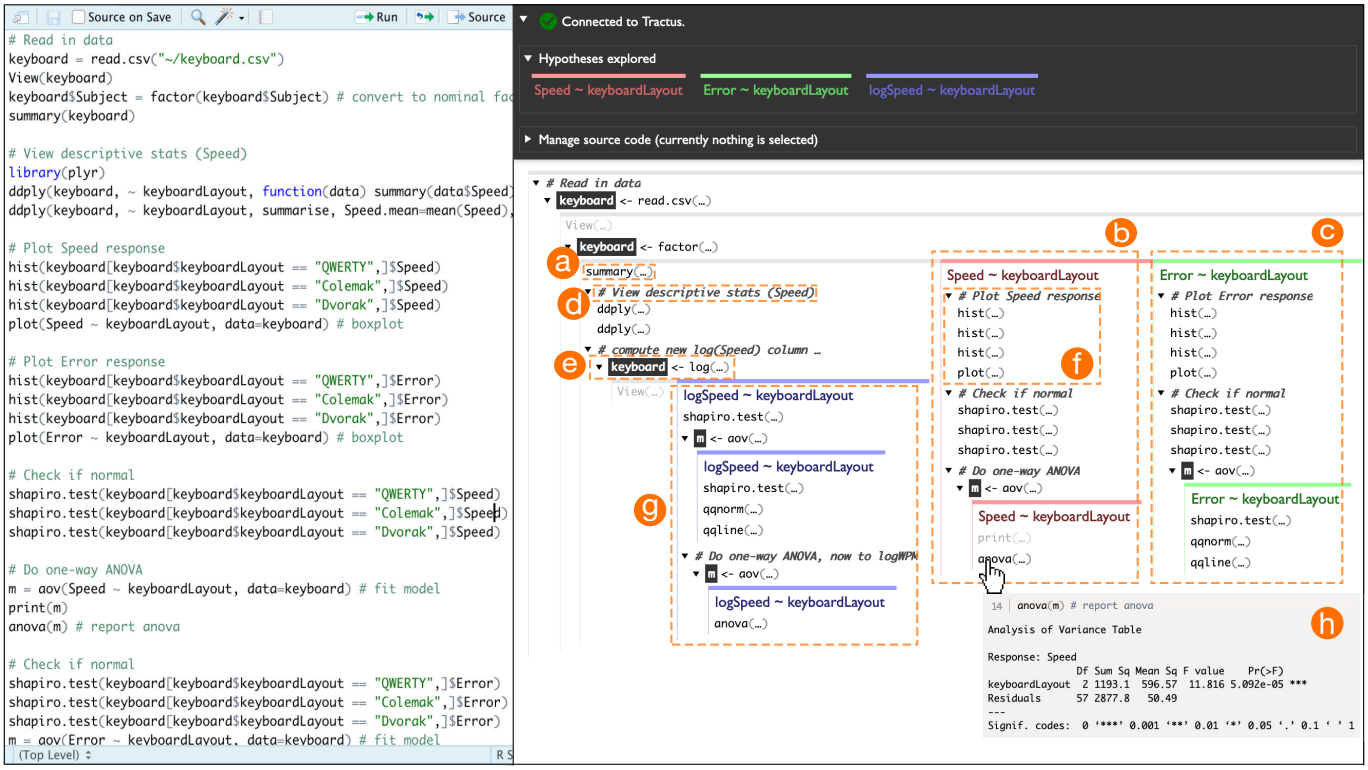

Figure 5. TRACTUS is an algorithmic and visualization extension to RStudio that can support data science workflows. TRACTUS detects, captures, and visualizes: (1) source code experiments grouped as hypotheses, e.g., Fig. 5c, (2) dependencies across source code, which are visualized as an indented branch in the tree, e.g., Fig. 5g shows code that is dependent on the log-transformed dataset `keyboard` (Fig. 5e), and (3) based on our formative study, information that data scientists use to recall rationale and insights such as *block* comments (Fig. 5d) and execution output (Fig. 5h). Code sections corresponding to hypotheses that have the same execution dependency are placed next to each other to facilitate comparison, e.g., Fig. 5b and 5c.

the ezANOVA function in the ezANOVA R package[5]. However, we found very few instances of this in our parser validation. We encountered false positives where the formula notation was used in a plotting function rather than for specifying relationships between variables. An example is the `ddply` function, in which the user uses the formula notation to specify how to split the data frame, e. g. `ddply(kbd, ~ Layout, function(data) summary(data$Speed))`. In general, however, we did not discover significant mismatches in our parser validation. In summary, the parser detects hypotheses by looking for the following:

- R's formula notations like `measurement ~ factor` and `measurement ~ factor1*factor2*factor3`. R's formula notations can be used to specify advanced factor designs.
- Dataset manipulation operations like subdivisions: `subset(data, factor == "level")$measurement`
- Dataset column selections, e.g., `data[data$factor == "level",]$measurement`

Capturing Code Blocks

Data scientists organize their source code into code blocks with a leading block comment (Finding #4). We wanted to capture such blocks; a block includes all statements in the block as well as the leading comment. To do so, whenever the parser encounters a line of code that is a comment, it assumes that a new block is present. All comments following the first line of comment are considered to be the block's comment until the first line containing an expression is encountered. This and all subsequent expressions are linked to the block until an empty line is encountered, upon which the block is closed.

Parser's Output: Hypothesis Tree

The parser uses a tree data structure to capture the hypothesis information of source code. We refer to this as the *hypothesis tree*. It is constructed by parsing the source code one statement at a time, extracting source code components and dependencies. To represent execution dependencies, the parser ensures that dependent statements are added as a child to the statements it depends on. (In situations where there are multiple parents, we pick the most recent parent in the source code to retain a tree structure. A DAG would reflect this one-to-many dependency more precisely, but our tree representation is simpler and resembles the *source code* more closely.) If the statement belongs to a hypothesis, it is added under the corresponding branch in the tree. (Each branch represents a hypothesis; a branch is created upon first encounter of a hypothesis in a statement.) Any metadata associated with the

[5] https://www.rdocumentation.org/packages/ez/versions/3.0-1/topics/ezANOVA

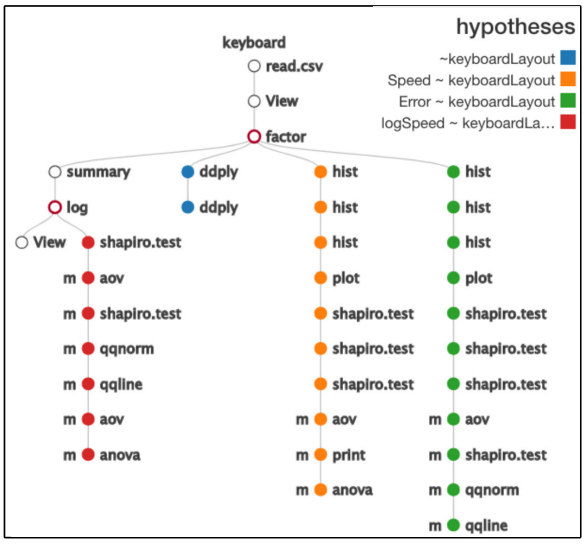

Figure 6. The first version of TRACTUS. After evaluating this version with users, we made several design improvements, e.g., symmetrical tree required horizontal scrolling for large files, and improved the underlying architecture in the current version.

statement, like its execution output and tail comment, is also added to the hypothesis tree.

Parser Evaluation

TRACTUS' parser was validated using a corpus of 38 R scripts, which were randomly sampled from the Open Science Framework (OSF)[6] and by solicitation from researchers at our local university. We were eventually able to achieve a 82.4% coverage with these files. 4 files had syntax errors and failed to execute. Of the remaining 34 files, TRACTUS successfully parses and visualizes 28 files. The parser failed to parse the remaining 6 files due to several reasons, e.g., deeply nested statements. (See supplements for details.) The parser can successfully parse large files ($>$ 7500 LOC). In such cases, the groupings in the visualization can be collapsed to aid navigation.

Visualization

The RStudio addin runs a web view alongside the R source code that visualizes the hypothesis tree. In this subsection, we describe the visualization and how users can interact with it. We start by describing the layout of the app, how information is presented and organized at a higher level of abstraction, and then discuss concrete details.

Layout

The visualization is shown next to the user's code. The top panel of the visualization provides an overview of the hypotheses explored in source code. Clicking on a hypothesis highlights the corresponding nodes in the visualization. The top panel also provides options to perform new explorations and generate code to reproduce results. (These features are discussed later in this section.) The rest of the visualization shows the user's code grouped into hypotheses.

[6] http://osf.io

Visualizing Dependencies and Hypotheses

TRACTUS aims to provide an overview of the user's work, and helps her transition from exploration to writing production code and reports. To support this, we chose a tree visualization instead of a graph as described earlier. Our algorithm constructs the tree visualization in the following manner:

1. If the statement has no dependencies, it is placed under the root node.
2. If the statement has one dependency, it becomes a child of the dependent statement's node in the tree, e.g., Fig. 5a.
3. If the statement has multiple dependencies, it becomes a child of the chronologically most recent parent.

The resulting visualization encodes the dependencies among statements. TRACTUS then uses the information about each statement's hypothesis for grouping. Statements that were determined to analyze a hypothesis are placed (e.g., Fig. 5c) under a branch and color coded. Consequently, statements that do not belong to a particular hypothesis, e.g., code used for loading datasets, are distinguishable from other code. As a second level of grouping, statements that belong to the same comment block are grouped, e.g., Fig. 5f. Unlike tail comments, block comments are explicitly shown to the user. Inside groups, statements retain their source code order.

Visualizing Contextual Information

To reduce visual clutter, TRACTUS progressively discloses [24] new information. For variable assignment statements, it displays only the variable and function names by default; additional information like the execution output, the statement's line number in an R code file, tail comment (if any), and the complete expression of the statement are revealed upon hovering with the mouse pointer, e.g., as shown in Fig. 5h. Users can collapse or expand branches in the visualization to focus on specific code groups—both at the level of hypotheses and code blocks.

TRACTUS uses visual cues to help users forage information faster. Prior statements that had changed a variable's value and statements that do not contribute to the business logic, e.g., `print()` and `cat()`, are displayed, but are intentionally made less noticeable.

Data Injection

To help semi-automate the exploration routine (Finding #2), TRACTUS supports data injection. This can help users who want to explore a new alternative based on an existing base code. The user selects the base code in the visualization, clicks on a button to inject data, and selects, from a list that TRACTUS creates by analyzing existing code, the measure and factor(s). TRACTUS then generates the code with new variables and copies it to the clipboard. This avoids the need to manually manage data dependencies.

Result Reproduction

TRACTUS can also generate code to reproduce the result of a statement. While this is not a novel feature [14, 37], it improves TRACTUS' utility. When the user selects one or more desirable statements, TRACTUS uses dependency information

to retrieve all statements necessary to reproduce the expected result.

Architecture

All components of TRACTUS can be modified independently of each other. This makes extensions easier, e.g., to work with more metadata, support more visual artifacts, or support other scripting languages like Python. The parser is written in Rust[7], a high performance, robust programming language, and returns a structured JSON tree that can be visualized differently if desired. The visualization is built using D3.js[8] and can be run in a web browser. The RStudio addin is written in R. For more implementation details, see supplements.

EVALUATION

We evaluated TRACTUS with users in two studies. The first explored how data scientists use TRACTUS to understand R code written by others. After using the results of this study to improve TRACTUS, in our second study we explored how TRACTUS helps data scientists in various stages of their analysis.

Study 1: Can TRACTUS Help Understand Source Code?

Three participants (1 female; 2 self-reported as intermediate users, 1 a beginner) used the initial version of TRACTUS (Fig. 6) to understand and then describe three R scripts. We sampled scripts of three different sizes (small: 25 LOC; large: over 500 LOC) from real-world research projects on OSF. Sessions were 40 minutes long on average.

Analysis and Findings

Our analysis motivated several design improvements. The symmetrical tree structure in this version required horizontal scrolling and was hard for the study participants to navigate, even for files that were only moderately long. Participants also mentioned that the visualization had too many details that added to the visual clutter. We fixed these issues and also improved the underlying architecture of TRACTUS to make it faster and more easily extendable.

All participants commented that TRACTUS helped them understand source code better than navigating code without TRACTUS, especially when the source code gets larger. P1 suggested better ways to group information in the visualization. P2 liked the hover-interaction, and mentioned that the visualization helped him easily spot which statistical model was used for each hypothesis.

Study 2: Can TRACTUS Improve Data Science Workflow?

We conducted a second study to validate the benefits of TRACTUS during experimentation and when writing production code. Seven academic data scientists (3 female, median age = 29) took part in the study. They were recruited through mailing lists and social media. P1, P5, and P6 self-identified as beginners, P2, P4, and P7 as intermediates, and P3 as an expert R analyst.

To establish a baseline of our participants' workflow, we asked them to first use RStudio *without* extensions before using RStudio with TRACTUS. Participants were given datasets[9] to analyze. Datasets had several measurements and factors; many hypotheses could potentially be validated from the dataset. To maintain ecological validity, participants were asked to first perform EDA to *generate hypotheses* by themselves, and then perform confirmatory analyses. Based on their findings, participants wrote a report of their work. After the analysis, participants gave their feedback about TRACTUS. All participants analyzed at least two datasets, and sessions were 100 minutes long on average.

Analysis and Findings

We analyzed the screen recordings by selectively theming the data [34] to identify the following:

Execution dependencies: P1, P3, P4, and P6 reported that the visualization of execution dependencies was useful during the initial exploratory phase. The visualization was particularly effective in helping participants track variables that were created a while ago. P3 compared the visualization to the Environment pane in RStudio, which is one approach used by participants to track variables when using RStudio without TRACTUS, mentioning that the ability to understand the origins of a variable was useful:

> *"[The execution dependency graph] reminds of the Environment pane, [but] it is just better as it [also] shows where [a] variable came from."* –P3

In this situation, the participant had not named the variable appropriately, but the dependency graph helped him infer the context (in this case, the variable was the result of a subset function).

Code curation and code quality: One unintended side effect of TRACTUS was that it encouraged participants to *curate* their code. After performing exploratory analysis, P2 and P3 used the visualization to remove scratchpad code from their script so that the visualization would become *less messy.* E.g., P2 found that there were several nodes in the visualization that represented his explorations to fix a bug; since this did not contribute towards the analysis, he wanted to delete these lines of code. P2 also mentioned that he would not have removed these lines of code when using RStudio without TRACTUS, indicating that the visualization improves *awareness* of source code. In contrast to removing source code, three participants (P3, P4, and P6) used the visualization to improve the quality of their R code, e.g., by renaming variables.

Exploration states: Since the visualization groups code according to hypotheses, it helped participants notice *patterns* across analyses. Several participants (P1, P3, P4, and P6) were able to compare the states of hypotheses to understand similarities and differences:

> *"[Using* TRACTUS*, it is] easier to compare analyses side by side to say 'yeah, it's the same' or find [out] what is different."* –P1

[7] https://www.rust-lang.org
[8] https://d3js.org
[9] Source: https://github.com/fivethirtyeight/data; see supplements for dataset details.

This also proved to be useful when writing reports later, since participants could easily detect differences between explorations.

Orientation and navigation: TRACTUS can help data scientists be more oriented during analysis. E.g., when analyzing his data, P2 wanted to test several hypotheses. He selected one and tested it, but while doing so, he identified another hypothesis and set off on a different analysis path. When this did not lead to promising results, P2 used TRACTUS to *backtrack* to the initial hypothesis to continue the analysis.

The benefits of TRACTUS do not cease after analysis. P4 mentioned that the visualization was useful to kick-start new analyses, since the visualization captures the analysis procedure more succinctly and is more easily understandable than source code.

Design improvements: We also identified several areas of improvement based on this study. Three participants (P1, P4, and P5) found the visual notation, especially execution dependencies, hard to understand initially. We redesigned the visualization to reduce clutter by reducing the information shown and by making some changes to the layout.

Overall, participants were mostly positive about TRACTUS and looked forward to using it. During all ten sessions, TRACTUS was able to detect the hypotheses accurately except for two instances. In both these instances, the participant specified the hypothesis in an unexpected manner, e.g.:

```
read.csv("~/data.csv")$measure ~
read.csv("~/data.csv")$factor
```

While this is valid, it is uncommon and our parser failed to detect the hypothesis. (The parser is programmed to only expect variables in a formula notation.)

DISCUSSION

Towards Reproducible, Transparent Data Science

For an analysis to be reproducible, executing its code should reproduce the expected, correct results. TRACTUS uses the execution dependencies to capture reproducible code. Additionally, TRACTUS can be extended to work with R packages like `reprex`[10], which provides more powerful sharing options. TRACTUS proposes a visualization that can help users get an overview of the analysis; this can be shared in research papers to promote transparency. Note that TRACTUS captures all source code in an R session, even the scratchpad code executed in the console. TRACTUS could be extended to capture Markdown[11] from R Notebooks, allowing more powerful narratives to be included in the visualization.

'Mindfully' Navigating the Garden of Forking Paths?

We believe that one of the prominent issues with NHST, wandering down the garden of forking paths, is a *design* problem. TRACTUS makes the paths (i.e., all analyses) visible to the data scientist. Our evaluation indicates that this improves user's awareness of source code, leading to code curation. This could be an antidote to over-testing, and help data scientists be more oriented and structured in their analysis. Additionally, TRACTUS can be extended to track all significance tests the data scientist conducts and warn against over-testing.

Extending the Approach

TRACTUS has the potential to be extended to other domains. Domains that use explicit notations for explorations (like the formula notation for hypothesis testing) can be accommodated. Other data science domains that do not fit this criteria, e.g., machine learning, would require a different method to detect explorations. Since this currently depends on syntactic signals in code, it could be programming language-specific.

Other programming languages used for hypothesis-driven data science, like Python, have syntax similar to R that can be leveraged to detect hypotheses. E.g,. Python uses the following syntax for selecting data: `variable = data[data['factor'] == level]['measure']`.

LIMITATIONS

TRACTUS parsed most (82.4%) of the R scripts it was tested with, as well as the scripts from our user studies. However, the parser does not support all R code; the supplements contain a list of limitations. Complex structures like deeply nested statements are supported, but slow down the parser significantly. Tail comments that occur before a statement or expression is complete, e.g., `for(i in 1:n) #Comment`, are not supported.

As mentioned earlier, participants from our motivational study were from academia. Thus, our findings may not generalize to all data scientists. E.g., data scientists in business may follow rigorous coding guidelines and write modular code, reducing problems with finding prior code and results. Also, in our motivational study, we observed data scientists for an hour. Real-world data science projects last weeks or longer, and the analysis code could span multiple files. TRACTUS currently does not support multiple files, but its underlying algorithm can be extended to do so with little effort.

SUMMARY

Data scientists produce valuable insights from data to influence our lives in profound ways. This paper discusses the problems they face in their work, and proposes TRACTUS to address these problems. Our qualitative evaluations show the benefits of TRACTUS. Among other benefits, its visualization can help read and understand existing analysis code, support new analyses, and serve as a lightweight medium to share analyses. TRACTUS is open source and is available at http://hci.rwth-aachen.de/tractus.

ACKNOWLEDGEMENTS

This project was partly funded by the German B-IT Foundation. We thank Ilya Zubarev for his contributions in the early stages of this project. We thank Christian Corsten, Nur Hamdan, Chat Wacharamanotham, and all reviewers for their valuable feedback. Finally, we thank all our participants for their time and involvement.

[10] https://github.com/tidyverse/reprex

[11] https://daringfireball.net/projects/markdown/

REFERENCES

[1] Ashraf Abdul, Jo Vermeulen, Danding Wang, Brian Y. Lim, and Mohan Kankanhalli. 2018. Trends and Trajectories for Explainable, Accountable and Intelligible Systems: An HCI Research Agenda. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 582, 18 pages. DOI: http://dx.doi.org/10.1145/3173574.3174156

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. *In Proceedings of the International Conference on Software Maintenance* (1998), 368–377. DOI: http://dx.doi.org/10.1109/icsm.1998.738528

[3] Robert Bogdan and Sari Knopp Biklen. 2007. *Qualitative Research for Education: An Introduction to Theory and Methods* (5 ed.). Pearson.

[4] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. 2010. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2503–2512. DOI: http://dx.doi.org/10.1145/1753326.1753706

[5] Paul Cairns. 2007. HCI... Not As It Should Be: Inferential Statistics in HCI Research. In *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...But Not As We Know It - Volume 1 (BCS-HCI '07)*. British Computer Society, Swinton, UK, UK, 195–201. http://dl.acm.org/citation.cfm?id=1531294.1531321

[6] Kwok Cheung and Jane Hunter. 2006. Provenance Explorer – Customized Provenance Views Using Semantic Inferencing. In *The Semantic Web - ISWC 2006*, Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora M. Aroyo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–227. DOI: http://dx.doi.org/10.1007/11926078_16

[7] Andy Cockburn, Carl Gutwin, and Alan Dix. 2018. HARK No More: On the Preregistration of CHI Experiments. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 141, 12 pages. DOI:http://dx.doi.org/10.1145/3173574.3173715

[8] Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. 2006. Code Thumbnails: Using Spatial Memory to Navigate Source Code. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC '06)*. IEEE Computer Society, USA, 11–18. DOI: http://dx.doi.org/10.1109/VLHCC.2006.14

[9] Pierre Dragicevic. 2016. *Fair Statistical Communication in HCI*. Springer International Publishing, Cham, Switzerland, 291–330. DOI: http://dx.doi.org/10.1007/978-3-319-26633-6_13

[10] Pierre Dragicevic, Yvonne Jansen, Abhraneel Sarma, Matthew Kay, and Fanny Chevalier. 2019. Increasing the Transparency of Research Papers with Explorable Multiverse Analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 65, 15 pages. DOI:http://dx.doi.org/10.1145/3290605.3300295

[11] Jean-Daniel Fekete, Danyel Fisher, Arnab Nandi, and Michael Sedlmair. 2019. Progressive Data Analysis and Visualization (Dagstuhl Seminar 18411). *Dagstuhl Reports* 8, 10 (2019), 1–40. DOI: http://dx.doi.org/10.4230/DagRep.8.10.1

[12] Andrew Gelman and Eric Loken. 2013. The Garden of Forking Paths: Why Multiple Comparisons Can Be a Problem, Even When There Is No "Fishing Expedition" or "p-Hacking" and the Research Hypothesis Was Posited Ahead of Time. *Department of Statistics, Columbia University* (2013).

[13] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance (TaPP'12)*. USENIX Association, Berkeley, CA, USA, 7. http://dl.acm.org/citation.cfm?id=2342875.2342882

[14] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 270, 12 pages. DOI: http://dx.doi.org/10.1145/3290605.3300500

[15] Yoshiki Higo and Shinji Kusumoto. 2009. Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *2009 16th Working Conference on Reverse Engineering*. IEEE Computer Society, 315–316. DOI:http://dx.doi.org/10.1109/WCRE.2009.39

[16] Macartan Humphreys, Raul Sanchez de la Sierra, and Peter van der Windt. 2013. Fishing, Commitment, and Communication: A Proposal for Comprehensive Nonbinding Research Registration. *Political Analysis* 21 (2013), 1–20. DOI: http://dx.doi.org/10.1093/pan/mps021

[17] Norbert L. Kerr. 1998. HARKing: Hypothesizing After the Results are Known. *Personality and Social Psychology Review* 2 (1998), 196–217. DOI: http://dx.doi.org/10.1207/s15327957pspr0203_4

[18] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1265–1276. DOI:http://dx.doi.org/10.1145/3025453.3025626

[19] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 92, 13 pages. DOI:http://dx.doi.org/10.1145/3290605.3300322

[20] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17)* (2017), 25–29. DOI: http://dx.doi.org/10.1109/vlhcc.2017.8103446

[21] Jan-Peter Krämer, Thorsten Karrer, Jonathan Diehl, and Jan Borchers. 2010. Stacksplorer: Understanding Dynamic Program Behavior. In *Adjunct Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology (UIST âĂŹ10)*. Association for Computing Machinery, New York, NY, USA, 433–434. DOI:http://dx.doi.org/10.1145/1866218.1866257

[22] Wendy L. Martinez and Angel R. Martinez. 2004. *Exploratory Data Analysis with MATLAB (Computer Science and Data Analysis)*. Chapman & Hall/CRC.

[23] Raymond S. Nickerson. 2000. Null Hypothesis Significance Testing: A Review of an Old and Continuing Controversy. *Psychological Methods* 5 (2000), 241. DOI: http://dx.doi.org/10.1037/1082-989x.5.2.241

[24] Jakob Nielsen. 2006. Progressive Disclosure. (2006). http://nngroup.com/articles/progressive-disclosure/ (last accessed on 20-09-2019).

[25] Gregory Piatetsky. 2018. How Many Data Scientists Are There? (2018). https://www.kdnuggets.com/2018/09/how-many-data-scientists-are-there.html (last accessed on 20-09-2019).

[26] Foster Provost and Tom Fawcett. 2013. Data Science and its Relationship to Big Data and Data-Driven Decision Making. *Big Data* 1 (2013), 51–59. DOI: http://dx.doi.org/10.1089/big.2013.1508

[27] Xiaoying Pu and Matthew Kay. 2018. The Garden of Forking Paths in Visualization: A Design Space for Reliable Exploratory Visual Analytics. *2018 IEEE Evaluation and Beyond - Methodological Approaches for Visualization (BELIV)* 00 (2018), 37–45. DOI: http://dx.doi.org/10.1109/beliv.2018.8634103

[28] D. W. Sandberg. 1988. Smalltalk and Exploratory Programming. *ACM SIGPLAN Notices* 23 (1988), 85–92. DOI:http://dx.doi.org/10.1145/51607.51614

[29] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology* 9, 10 (10 2013), 1–4. DOI: http://dx.doi.org/10.1371/journal.pcbi.1003285

[30] Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. 2011. False-Positive Psychology: Undisclosed Flexibility in Data Collection and Analysis Allows Presenting Anything as Significant. *Psychological Science* 22 (2011), 1359–1366. DOI: http://dx.doi.org/10.1177/0956797611417632

[31] Krishna Subramanian, Johannes Maas, Michael Ellers, Chat Wacharamanotham, Simon Voelker, and Jan Borchers. 2018. StatWire: Visual Flow-based Statistical Programming. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems (CHI EA '18)*. ACM, New York, NY, USA, Article LBW104, 6 pages. DOI: http://dx.doi.org/10.1145/3170427.3188528

[32] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual Augmentation of Source Code Editors: A Systematic Mapping Study. *Journal of Visual Languages & Computing* 49 (2018), 46–59. DOI: http://dx.doi.org/10.1016/j.jvlc.2018.10.001

[33] John W Tukey. 1980. We Need Both Exploratory and Confirmatory. *The American Statistician* 34 (1980), 23–25. DOI: http://dx.doi.org/10.1080/00031305.1980.10482706

[34] Max van Manen. 1990. Beyond Assumptions: Shifting the Limits of Action Research. *Theory Into Practice* 29, 3 (1990), 152–157. http://www.jstor.org/stable/1476917

[35] Michael L. Van De Vanter. 2002. The Documentary Structure of Source Code. *Information and Software Technology* 44, 13 (2002), 767–782. DOI: http://dx.doi.org/10.1016/s0950-5849(02)00103-9 Special Issue on Source Code Analysis and Manipulation (SCAM).

[36] Rajesh Vikraman. 2018. Global Report on State of Data Science & Machine Learning - 2018 Based on Kaggle Survey. (2018). https://rpubs.com/cvrajesh/kagglesurvey2018 (last accessed on 20-09-2019).

[37] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[38] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. 2016. Chronicler: Interactive Exploration of Source Code History. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 3522–3532. DOI:http://dx.doi.org/10.1145/2858036.2858442

[39] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of Fine-Grained Code Change History. *2013 IEEE Symposium on Visual Languages and Human Centric Computing* (2013), 119–126. DOI: http://dx.doi.org/10.1109/vlhcc.2013.6645254