Supporting Data Workers To Perform Exploratory Programming

Krishna Subramanian
RWTH Aachen University
52056 Aachen, Germany
krishna@cs.rwth-aachen.de

Ilya Zubarev
RWTH Aachen University
52056 Aachen, Germany
ilya.zubarev@rwth-aachen.de

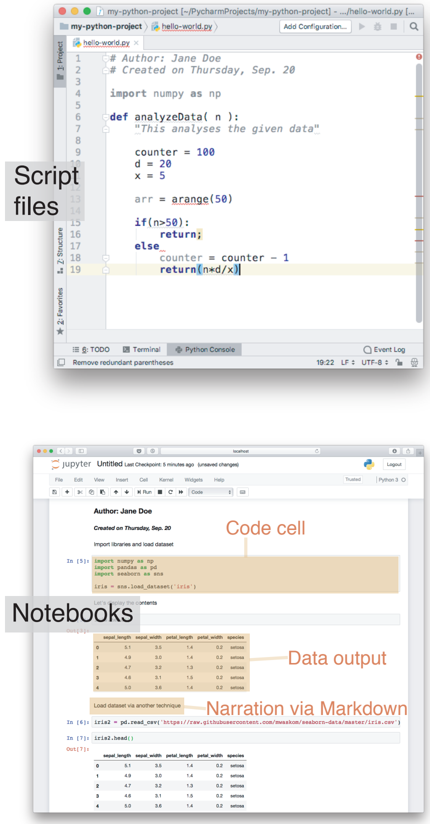
Simon Völker
RWTH Aachen University
52056 Aachen, Germany
voelker@cs.rwth-aachen.de

Jan Borchers
RWTH Aachen University
52056 Aachen, Germany
borchers@cs.rwth-aachen.de

ABSTRACT

Data science is an open-ended task in which *exploratory programming* is a common practice. Data workers often need faster and easier ways to explore alternative approaches to obtain insights from data, which frequently compromises code quality. To understand how well current IDEs support this exploratory workflow, we conducted an observational study with 19 data workers. In this paper, we present two significant findings from our analysis that highlight issues faced by data workers: (a) *code hoarding* and (b) excessive *task switching* and *code cloning*. To mitigate these issues, we provide design recommendations based on existing work, and propose to augment IDEs with an interactive visual plugin. This plugin parses source code to identify and visualize high-level task details. Data workers can use the resulting visualization to better understand and navigate the source code. As a realization of this idea, we present *HypothesisManager*, an add-in for RStudio that identifies and visualizes the hypotheses that a data worker is testing for statistical significance through her source code.

CHI'19 Extended Abstracts, May 4–9, 2019, Glasgow, Scotland UK
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5971-9/19/05.
https://doi.org/10.1145/3290607.3313027



```
# Author: Jane Doe
# Created on Thursday, Sep. 20

import numpy as np

def analyzeData( n ):
    "This analyses the given data"

    counter = 100
    d = 20
    x = 5

    arr = arange(50)

    if(n>50):
        return;
    else
        counter = counter - 1
        return(n*d/x)
```

```
Remove redundant parentheses                19:22  LF  UTF-8
```

Author: Jane Doe

Created on Thursday, Sep. 20

Import libraries and load dataset

```
In [5]: import numpy as np
        import pandas as pd
        import seaborn as sns

        iris = sns.load_dataset('iris')
```

Let's display the contents

```
Out[3]:
   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1          3.5           1.4          0.2   setosa
1           4.9          3.0           1.4          0.2   setosa
2           4.7          3.2           1.3          0.2   setosa
3           4.6          3.1           1.5          0.2   setosa
4           5.0          3.6           1.4          0.2   setosa
```

Load dataset via another technique

```
In [6]: iris2 = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv')

In [7]: iris2.head()
Out[7]:
   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1          3.5           1.4          0.2   setosa
1           4.9          3.0           1.4          0.2   setosa
2           4.7          3.2           1.3          0.2   setosa
3           4.6          3.1           1.5          0.2   setosa
4           5.0          3.6           1.4          0.2   setosa
```

Figure 1: Scripting language IDEs allow data workers to write code using two main modalities. Script files (above) allow for traditional code storage and execution. Notebooks (below) allow programmers to organize code into cells, show data and graphs alongside code cells, and include narratives via Markdown.

KEYWORDS

Scripting languages, Exploratory programming, Grounded Theory methodology, Programming interfaces, Data Science

INTRODUCTION

Data science uses quantitative and qualitative methods to solve relevant problems and predict outcomes from data. It is a frequent activity for data workers in both industry and academia [6], and includes tasks such as statistical machine learning and significance testing. Data science often has open-ended goals: Data workers often explore different approaches to obtain insights from data, and their goal evolves based on these insights. This exploratory workflow is called *exploratory programming* [4]. During this activity, data workers often compromise the quality of their source code, in order to explore alternatives more quickly *(quick prototyping)*. Thus, the needs of data workers performing exploratory programming are different from traditional programming.

Scripting languages like Python and R are commonly used by data workers for exploratory programming [8]. However, the IDEs for these scripting languages follow the interaction design of traditional programming IDEs, which may not suit the exploratory workflow of data science. We therefore wanted to understand how well current scripting language IDEs support exploratory programming. We collected and qualitatively evaluated data from 19 data workers who use scripting languages for data science. We discuss two significant issues from our analysis that have design implications for future IDEs: *code hoarding*, and excessive *task switching* and *code cloning*. To address these issues, we present design recommendations from existing HCI research and propose to augment IDEs with an interactive visual plugin.

RELATED WORK

Kery et al. and Rule et al. have investigated how data workers use interactive Python notebooks [5, 9]. Notebooks allow data workers to weave narrative with source code, which is organized into cells, and to juxtapose output and graphics with code. Their research showed that data workers consider notebooks to be personal and messy, and that data workers have difficulties integrating version control and sharing notebooks with others. In another study, Kery et al. investigated how data workers perform exploratory programming in general, not with notebooks in particular [3]. They conducted semi-structured interviews with 10 researchers, and discovered that version control needs to be better integrated into current IDEs. Unlike existing research, our study is based on observational data, walkthroughs, and files or notebooks written by data workers, in addition to interviews. We also accounted for variability in data workers' domains, expertise, and scripting languages.

Table 1: Experience (in years), domain of the task, and scripting language(s) used by participants in our study.

ID	Experience in years	Domain *(Scripting Language)*
*P*01	1	ST (*R*)
*P*02	2	ST (*R*)
*P*03	1	ST (*R*)
*P*04	1	ST (*R*)
*P*05	2	ML (*Python*)
*P*06	5	3D (*Python*)
*P*07	3	ST (*R*)
*P*08	2	ML (*Python*)
*P*09	0.5	FA (*R*)
*P*10	3	ML (*Python*), ST (*R*)
*P*11	1	3D (*Python*), NA (*MATLAB*)
*P*12	2	EM (*R*), ST (*R*)
*P*13	1	ML (*Python*), ST (*Python*, *R*)
*P*14	5	ML (*Python*, *MATLAB*)
*P*15	3	ML (*Python*)
*P*16	10	ML (*Python*, R)
*P*17	3	ML (*Python*, *MATLAB*)
*P*18	7	NA (*MATLAB*)
*P*19	8	NA (*MATLAB*, *Python*)

ST: Significance testing
ML: Machine learning
3D: 3d modeling
FA: Financial analysis
NA: Numerical analysis
EM: Equation modeling

METHODOLOGY

To obtain an unbiased understanding of how data workers use scripting language IDEs, we followed the Constant Comparative Method [1] of the Grounded Theory methodology [10]. We collected data from 19 data workers (8 female) with different backgrounds, task domains, and scripting languages used. Data was collected through interviews (30 min. on average, all 19 participants), observations (40 min. on average, 12 participants) of participants working on a real-world (n=7) or fabricated task (n=5), and 41 script files/notebooks authored by participants. Through two cycles of coding, we iteratively generated our findings by constantly comparing them with the data collected. After several iterations of initial coding, the first author developed a coding guide, which was used by an independent coder on two interview transcripts and two observational videos to achieve a Cohen's Kappa of $\kappa = 0.84$. For details about participants' experience, domains, and scripting languages used, see Table 1.

FINDINGS

Code Hoarding

Since data science involves exploring multiple alternatives, data workers need to keep track of several code chunks in parallel. These code chunks are stored either within or across script files and/or notebooks. When the data worker explores an alternative, she does not know if the approach will pan out to be successful. Therefore, she cannot preemptively decide how to organize code chunks. Furthermore, data science tasks do not always have an optimal solution—data workers often pick sub-optimal solutions, e.g., due to performance considerations. This lack of confidence in knowing whether a piece of code will be used later or not leads to the data workers being *cautious of losing code*. This cautiousness in turn leads to *code hoarding*: Data workers become reluctant to delete analysis code in script files and interactive notebooks. E.g., *P*12 had several script files named `Untitled` stored on his local disk. Although he was not sure what these script files were about, he did not want to delete them for fear of losing something valuable. Some data workers find it difficult to clean up their notebook or script files, and simply do not want to make the effort because they have already obtained the insights from the data. This occurred more prominently in notebooks, which do not provide a standard way of structured use.

Consequences: In addition to memory considerations, code hoarding can lead to an overload of relevant code alternatives that the data worker needs to keep track of. In extreme situations, data workers may retain code that does not even execute anymore:

> *"I store everything in notebooks—even the explorations with fake data. I don't clean it, since I might need it in the future … most code does not even execute now, but I just let it be.*[1]*" - P13*

[1]Paraphrased for better readability.

Excessive Task Switching and Code Cloning

Another theme that emerged from our analysis is that exploratory programming leads to excessive task switching and code cloning. While these issues are not uncommon for traditional programming, they were more pronounced for exploratory programming.

Data science does not occur in isolation—during or after exploration of alternatives, data workers need to document their progress for dissemination, e.g., by writing a publication or producing a report. This usually also involves moving the code, often from a notebook to a script file, and refactoring it. There are *two major implications* of this: Data workers constantly switch between applications or modalities, and code chunks are often moved across multiple applications, resulting in clones.

Regarding use of multiple applications and modalities, we found that most users who used notebooks also used script files. These users preferred interactive notebooks to explore alternatives, but script files to store their code, because script files allowed for easier testing, execution from the bash shell, and, in the case of machine learning tasks, compatibility with powerful compute clusters to run on. However, users found the need for both notebooks and script files frustrating:

> *"Sometimes I wonder if I can do my task entirely in the notebooks. This way, I don't have to copy-paste source code to script files, which is annoying. But then I have to make sure I properly test it and make really sure that my code works, which would take time.[2]" - P17*

[2]Paraphrased for better readability.

Note that although some notebook environments allow data workers to export code cells to script files, data workers reported often having to make modifications afterwards anyway.

Consequences: Data workers lose their mental model of the task as a result of excessive task switching; they need more time to read and understand a piece of code, whether it resides in a script file or interactive notebooks. Consequently, data workers also need more time to decide whether a given code segment implements an approach they want to go ahead with. Excessive cloning further lessens the readability of code and makes source code maintenance more difficult.

DESIGN RECOMMENDATIONS

The main cause of code hoarding is the data workers' fear of losing something valuable. Adding more explanations, e.g., by following the recommendations in [9], could help data workers understand code better, and therefore better decide if it is still relevant. To help decide the relevance of various approaches, data workers could tag their code chunks or cells in interactive notebooks (e.g., as in [3]). Automatically generated source code visualizations (e.g., [2]) could help with excessive task switching.

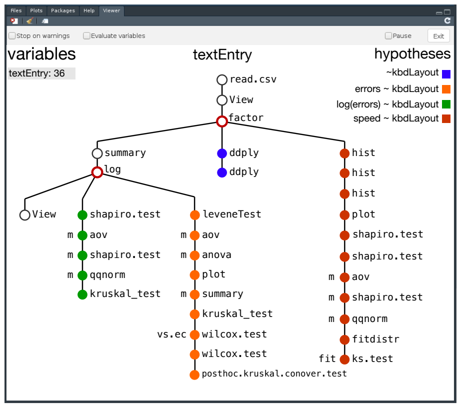


Figure 2: *HypothesisManager* is an add-in for RStudio that visualizes R source code as groups of hypotheses. It is shown alongside the native source code editor in RStudio and can be used to navigate through the source code file and inject data.

HYPOTHESIS MANAGER

This section describes our proof-of-concept system, *HypothesisManager*, that can be used to better support explorations in statistical significance testing tasks. The add-in is integrated into the RStudio GUI and is shown alongside source code in the 'Viewer' pane. A parsing engine detects the variables in the source code and attempts to identify the datasets and column names. Function call stacks of code expressions and the R formula notation help identify hypotheses. If the result of such an expression gets assigned to a variable, this variable is classified as a *statistical model*, and is tracked throughout the script to track the hypothesis.

The *source code model*, which refers to the information that will be visualized, consists of three types of entities: *variables*, *functions*, and *hypotheses*. Variables are the backbone of the visualization, with the relevant function calls represented hierarchically to show how the analysis evolves as different hypotheses are explored.

Interaction: Data workers navigate source code by clicking on the function in the visualization. Due to space constraints, we only visualize the parent function in a statement that has nested functions. For assignment operations in which a statistical model is assigned to a variable, and for subsequent use of this variable in a function invocation, we denote both the variable name and the parent function. The complete line of code in the source file, user's comments for this line of code to help the user recall her thought process, and links to the R help pages for the functions invoked in this line of code are shown as tooltips. Data workers can use *HypothesisManager* while performing statistical significance testing in RStudio and while viewing existing R script files written by themselves or others.

Data Injection: Since *HypothesisManager* can determine the dataset model (measurements and factors), it can also *inject* data into a current hypothesis. Data workers can avoid having to manually find and replace variable names in the source code. This could allow for faster explorations.

Design Process and Validation: We developed *HypothesisManager* iteratively, progressively developing our ideas from paper sketches. While we initially considered a variable-based grouping, we discovered that data workers actually think about their analysis in terms of hypotheses. We validated the parsing engine of *HypothesisManager* with a corpus of 40 R scripts that were randomly sampled from OSF[3] and from researchers at our local university. We eventually achieved a 75% coverage.

Applicability to Other Domains: While our parsing engine currently works only for significance testing, it can be extended to other domains like machine learning. This can be done by first identifying what constitutes a 'hypothesis' in this domain and then classifying the identified functions accordingly. E.g., a classification pipeline would constitute *parsing*, *attribute generation*, *parsing*, and *testing* steps [7].

[3]http://osf.io

SUMMARY

Code hoarding, and excessive task switching and code cloning are issues that plague data workers when performing exploratory programming tasks. We discussed existing HCI research as well as our proof-of-concept system *HypothesisManager* as possible solutions for these issues. While we believe that *HypothesisManager* has potential benefits, user studies are required for validation. We are working on making *HypothesisManager* available to data workers as open source, see http://hci.rwth-aachen.de/r-hypothesis-manager.

ACKNOWLEDGMENTS

This work was funded in part by the German B-IT Foundation. We thank all participants in our study for their insights. We also thank Christian Corsten, Nur Hamdan, and Prof. Janne Lindqvist for their feedback.

REFERENCES

[1] Hennie Boeije. 2002. A Purposeful Approach to the Constant Comparative Method in the Analysis of Qualitative Interviews. *Quality and Quantity* 36, 4 (01 Nov 2002), 391–409. https://doi.org/10.1023/A:1020909529486
[2] 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 *(CHI '10)*. ACM, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706
[3] 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. https://doi.org/10.1145/3025453.3025626
[4] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (VL/HCC '17)*. IEEE, 25–29. https://doi.org/10.1109/VLHCC.2017.8103446
[5] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 174:1–174:11. https://doi.org/10.1145/3173574.3173748
[6] Cathy O'Neil and Rachel Schutt. 2013. *Doing Data Science: Straight Talk From The Frontline*. O'Reilly Media, Inc.
[7] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. 2010. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1866029.1866038
[8] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. 2011. A Survey of the Practice of Computational Science. In *State of the Practice Reports (SC '11)*. ACM, New York, NY, USA, Article 19, 12 pages. https://doi.org/10.1145/2063348.2063374
[9] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 32, 12 pages. https://doi.org/10.1145/3173574.3173606
[10] Anselm Strauss and Juliet Corbin. 1994. Grounded Theory Methodology: An Overview. *Handbook of Qualitative Research* 17 (1994), 273–85.