Coding Be eR: Assessing and Improving the Reproducibility of R-Based Research With containR

Thesis advisor: Margo Seltzer and Joe Blitzstein Christopher Chen

Coding BetteR: Assessing and Improving the Reproducibility of R-Based Research with containR

Abstract

Reproducibility is the cornerstone of science, and we are in the midst of a reproducibility crisis. Simply sharing the code and data used for obtaining results is often insufficient for reproducibility; in fact, we show that 85.6% of the thousands of R programs published on Dataverse [1] since 2015 cannot be run. Moreover, our finding that the failure rate of these published R programs holds constant regardless of their age implies that errors are caused by code incorrectness, not age-related incompatibility. We contribute to the reproducibility of R-based research by building tools to both automatically correct common errors found in published code/data archives and package the archives to guarantee future reproducibility. We motivate developing these tools with analyses showing that only three types of mistakes caused more than 70% of all the errors we observed, and that automatically correcting these mistakes frequently revealed a more fundamental error: many datasets were simply missing the data used for analysis, highlighting the need for a better system of documenting and including research-code dependencies. We provide an example of such a system by building `containR`, a web application which combines our automatic error-correcting code and existing dependency detection tools to create easily-executable and platform-agnostic archives of R-based research.

[1] a web platform specifically designed for the open sharing of data and code used for result generation in published research

Contents

Listing of figures

To the memory of my mother, Kaiyan Jiang.

Acknowledgments

I would first like to thank my advisers, Margo Seltzer and Joe Blitzstein, for their expert guidance, unrivaled vision, and endless patience. You made my first foray into real academic research deeply enjoyable.

I would also like to thank my primary research collaborators, Matt Lau and Thomas Pasquier, for their selfless dedication toward helping me think through my research questions and providing support with provenance-based tools, as well as Aaron Ellison, who, without ever meeting me, instantly agreed to read this thesis.

Finally, I could not have completed this thesis without the endless love and support of my friends and family, including Jian Chen, Xi Chen, Scarlett Cheon, Phillip Huang, Harry Xue, Josh Felizardo, Scott Sun, Raghu Dhara (who is a web infrastructure wizard), and too many more to mention.

1 Introduction

1.1 Motivation

Experimental results of scientific research must be repeatable, not merely available. Thus, the well-documented reproducibility crisis facing data-driven scientific research [1] more than hinders scientific progress: it threatens the legitimacy of the entire scientific process. However, scientists are not unarmed in combating this crisis; data provenance has emerged as a promising tool to aid them in the fight. Introducing data provenance, Pasquier et al. wrote:

> Provenance was originally a formal set of documents to describe the origin and ownership history of a work of art. These documents are used to guide the assessment of the authenticity and quality of the item. In a computing context, data provenance represents, in a formal manner, the relationships between data items (entities),

> transformations applied to that data (activities), and persons or organizations associated with the data and transformations (agents). It can be understood as the record of the origin and transformations of data within a system. [2]

More concretely, data provenance describes computational procedures using graphs, with nodes representing data items, data-transforming activities, or people and organizations, and edges representing the temporal and causal relationships between nodes. By encoding research programs' dependencies and logic in a formal, well-ordered manner, these graphs legitimize research results and facilitate their regeneration. However, since data-driven analyses have a tendency to heavily leverage computational resources, manually creating provenance graphs for even a short body of research code would be extremely impractical, if not impossible. Therefore, there have been significant efforts [2–8] to create tools for automatically generating data provenance. These tools minimize required effort from the researcher by capturing provenance data at execution time and storing it for posterity using a standardized provenance data model created by the World Wide Web Consortium.[1] A number of tools have focused specifically on native provenance capture for the R programming language [3–5].

Because of its extensive modeling and data visualization functionality and its interpretive environment which allows users to interact with previously executed commands [9], R enjoys extensive use in data-driven scientific research spanning numerous fields outside its original area of application for statistics, and based on web-traffic statistics from the popular online debugging forum, Stack Overflow, R's popularity is rapidly increasing compared to that of other languages (see Fig. 1.1.1).[2] However, the very same interpretive environment which makes R so popular for research causes further problems for repeating said research. Pasquier et al. explained:

[1] https://www.w3.org/TR/prov-dm/
[2] https://stackoverflow.blog/2017/10/10/impressive-growth-r/

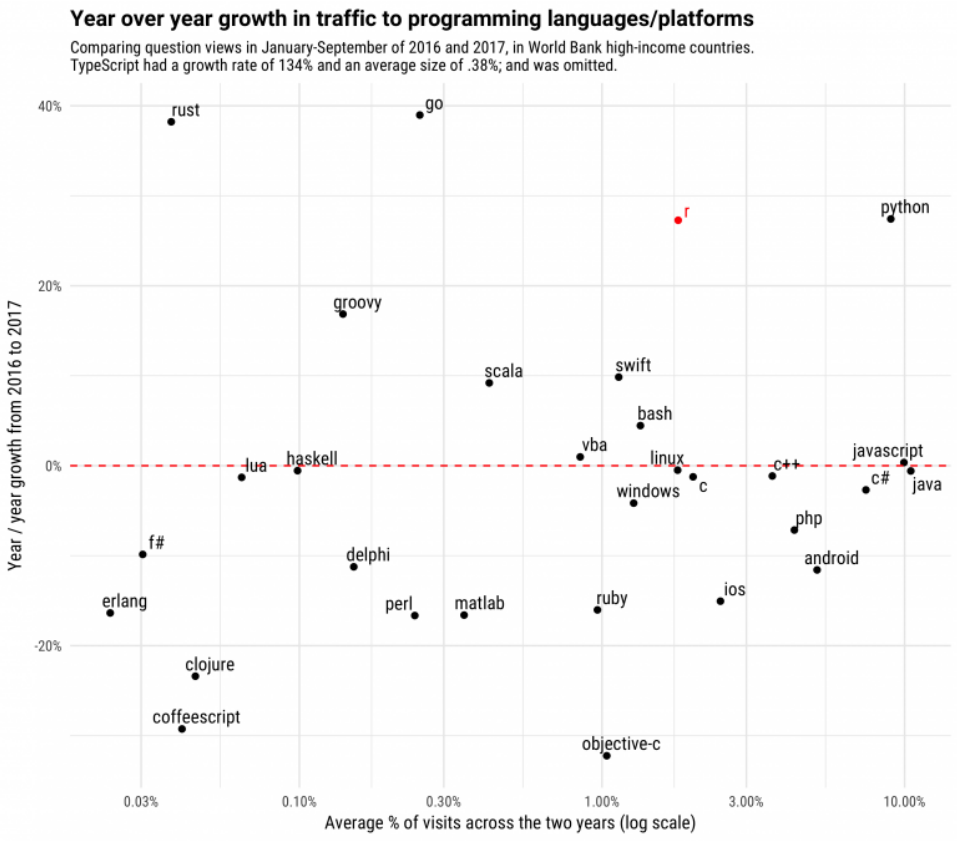

Figure 1.1.1: R has demonstrated impressive levels of growth in the past year. Taken from https://stackoverflow.blog/2017/10/10/impressive-growth-r/

> [R and other scripting languages] have [in addition to facilitating data-driven approaches] also enabled scientists to produce code of relatively mediocre quality, as their goal is the production of scientific insight via statistical results and graphics, not robust code. There are few incentives for a researcher to take the time to properly structure, annotate, and then curate the code to improve its legibility. Thus, data and code are prevalent, but much of it is either no longer usable or unintelligible. [4]

The tendency for R-using researchers to write non-robust code combined with R's increasing popularity make R the perfect candidate for applying automatic provenance capture tools. Existing tools, including `encapsulator` [4], `RDataTracker` [3], and `ProvR` [5], all try to impose minimal overhead for the scientist writing the R code. However, since they capture provenance at the time of code execution, these tools inherently assume that the researchers' R scripts will run error-free until completion, and fail to capture provenance when this assumption is false. Given the aforementioned concerns about researchers lacking the incentives and training to write robust R programs [4], this assumption may prevent automated provenance collection in R from working in its most-needed cases. In order to further improve the reproducibility of R-based research, we must therefore investigate the common causes of errors in researchers' R programs, and supplement existing tools that capture and apply data provenance with tools to preemptively address these errors. Since investigating common error sources necessitates running a large volume of R code, our investigative process also involves a case study of reproducibility in R-based research.

1.2 Related Work

To motivate our attempts to improve the reproducibility of computational research, we cite a recent study by Stodden et al. which demonstrates the sore need for this improvement. Stodden et al. assessed the reproducibility of results from a random sample of 204 scientific papers published in the journal *Science* after February 2011, when the journal implemented a policy designed to improve reproducibility by requiring researchers to fulfill all reasonable requests for the data and code needed to generate their results [1]. Despite this policy, Stodden et al. were only able to obtain data and code for 89 articles in their sample of 204 (about 44%), and of those 89 that provided data and code, deemed that results could only be potentially reproduced for 56 articles (about 27%). Their methodology for determining potential reproducibility allowed for significant

effort on the part of the result-verifying researcher:

> We noted where documentation on installing and running available codes was missing from both the paper and the location of the software, and continued with our best attempts at reproduction. We noted the cases where scripts and parameter files were missing. If the missing files could be recreated with some reasonable amount of effort (<100 lines of code) based on the information provided in the compendia, this was attempted. [1]

As such, their 27% finding can be thought of as a generous upper bound of the actual rate of computational reproducibility. The generosity of this upper bound only increases if we consider how *Science* had already implemented a policy specifically designed to address this issue. The small sample size of article results they actually attempted to replicate (only 22) and the generous use of researcher time in attempting replication leave room for a more strictly-defined case study of computational reproducibility. In addition to finding the reproducibility rate, Stodden et al. also noted that the "lack of standards for documentation and metadata for data, code, and workflows" seriously hindered their reproducibility efforts, suggesting the need for a more formalized metadata system for computational research such as that provided by data provenance [1].

Even the most motivated research cannot progress without the correct tools, and ours depends heavily on existing projects to capture data provenance natively in R. The `RDataTracker` project represents the first attempt to do so [3]. As a native R package (installable and runnable from within R), `RDataTracker` dramatically improved the convenience of provenance collection for R users by allowing them to control provenance capture from within a standard R session, whereas earlier non-native tools like Vistrails [8], Kepler [6], and Taverna [7] forced researchers to substantially alter their workflows. `RDataTracker` also improves on earlier, completely automatic tools by allowing users more control over the specific provenance collected; while automatic tools capture provenance for all aspects of the user's current R session, `RDataTracker` allows users to

specify start and end points with more granularity from within an R session, thus simultaneously reducing the volume and increasing the relevance of the collected data. `ProvR` is a simplified fork, or alternative version, of `RDataTracker`, which keeps `RDataTracker`'s provenance capturing functionality while presenting users with a cleaner interface of functions for controlling the capture process [5].

While provenance collection provides a formal record of computation, one would probably have difficulty finding a researcher pedantic enough to manually inspect the provenance graph for each script they write or use. The actual data provided by data provenance may therefore appear pointless at first glance. `encapsulator` and `Rclean` prove otherwise by leveraging native R provenance capture through `RDataTracker` to implement additional functionality for researchers: respectively, the creation of downloadable "capsules" containing all the code and data dependencies necessary to reproduce a research result [4] and the cleaning of originally-messy user-provided R source code files based on isolating code for producing specific outputs [10].

These tools for capturing provenance in R and using captured provenance to provide additional functionality for researchers are indisputably innovative. But, because they have not yet seen widespread usage in the R community, their robustness and performance while ingesting actual research code was untested. Additionally, even though these packages can be run with a few commands, all of them inevitably require interested researchers to make a concerted effort to install the packages' dependencies and learn package-specific syntax, imposing additional overhead which may dissuade researchers from voluntarily adopting these packages into their workflows.

1.3 Our Contributions

Our research had two high-level aims: (1) To perform a comprehensive assessment of the state of computational reproducibility in R, including stress-testing existing provenance capture tools; (2) To further improve the reproducibility of R-based research by modifying or creating provenance-based

tools in response to the results from our first aim.

When considering data sources to complete our assessment, we found Harvard University's installation of Dataverse [11] to be the most convenient and relevant option. Dataverse is a web application self-described as:

> an open source web application to share, preserve, cite, explore, and analyze research data. It facilitates making data available to others, and allows you to replicate others' work more easily. Researchers, data authors, publishers, data distributors, and affiliated institutions all receive academic credit and web visibility. [11]

To collect our data, we used Python to interface with the Dataverse API [3] and downloaded the datasets (consisting of both code and data) for 810 different R-based studies. We used a combination of scripts written in R and bash to attempt the execution of researchers' R programs, capturing provenance if the research programs successfully ran to completion and capturing the first line of error output if they failed. To complete this analysis in a timely fashion, we harnessed the computing power of Harvard University's Odyssey Computing Cluster [4] to parallelize our workflow by running the code files for each study simultaneously in a separate job, or scheduled computational task. We then aggregated the execution data for all the downloaded studies in order to analyze overall error occurrences across the R-based datasets.

Using the results of our reproducibility assessment, we wrote functions in Python to identify and correct the most commonly-observed errors in the R programs by directly parsing and editing their source code. With `containR`, we integrated our R-code correction functionality, automatic provenance collection with `ProvR`, and Docker[5] (a platform which allows an application to run efficiently on all major operating systems) into a simple web application to expedite the reproducibility of R-based research with minimal overhead for researchers.

[3] http://guides.dataverse.org/en/latest/api/index.html
[4] https://www.rc.fas.harvard.edu/odyssey/
[5] https://www.docker.com/get-docker

1.4 Outline

In Chapter 2 we detail the process of collecting R-code-containing datasets from Dataverse and the pre-processing we performed on the data. The process of downloading the datasets using Odyssey required three nested layers of Dataverse API calls, and did not work on Odyssey until a few weeks before the time of writing because of networking errors. We designed our analysis scripts to fully exploit the parallel computing capacity of Odyssey.

In Chapter 3, we describe and analyze the results from running the researchers' code. We describe the most common errors we obtained, our interpretation of the errors, and the relationships between error occurrences and the year and subject of the datasets. We also describe the approaches we took to automatically fix the most common errors and the results of attempting to execute the R scripts after we applied our fixes as a pre-processing step.

In Chapter 4, we discuss the functionality and implementation-level details of the `containR` web application. We initially intended on building a web interface for `encapsulator`, but after some testing, concluded that it would be unwieldy to run `encapsulator` simultaneously for multiple users in a web setting. Instead, we implemented `encapsulator`-like functionality using Python, `ProvR`, and Docker. We then walk through the features of `containR`. For each feature, we provide an example of its correct functionality followed by a more-detailed description of the methods and dependencies used for its implementation.

2

Data Collection and Preprocessing

2.1 Downloading Datasets from Dataverse

Before we could examine the reproducibility of R-based research, we needed to secure a large collection of R scripts used for research along with their corresponding data dependencies. We first considered reaching out to researchers over email and asking for their code and data, similar to Stodden et al. [1], but quickly decided against it over scalability and human fallibility concerns. With over 50,000 datasets, and a mission to encourage research data visibility and reproducibility, Dataverse[1] seemed the ideal source for legitimate, published datasets. However, despite the large number of datasets overall, it was unclear what proportion of datasets contained R-based analyses. Using the Python

[1] https://dataverse.org/

`requests` module and file-type query syntax,[2] we queried the Harvard Dataverse Search API[3] for the metadata of all published R programs, then parsed the metadata using the Python `json` and `re` (regular expression) modules to compile a list of DOIs (persistent digital object identifiers for intellectual property guaranteed to be unique to each dataset). Upon finding that this resultant list contained 729 unique DOIs, we decided that this data collection option was worth pursuing further. At the time of writing, this number has increased to 846 unique DOIs, whose datasets contain 3,407 unique R scripts.

For downloading files, Dataverse provides the Data Access API. [4] However, there was no way to programmatically download entire datasets from Dataverse.[5] Thus, to download each dataset DOI, we used the Python `requests` module to query the Dataverse Native API [6] for IDs of all files in the DOI's dataset, then used this DOI-to-file-ID mapping to query the Data Access API for each individual files within each dataset (see Figure 2.1.1).

We used Harvard University's Odyssey Cluster to parallelize each dataset download in a separate job with near-complete lack of concern for storage capacity (each research group on Odyssey receives a shared storage volume with a 50 terabyte limit.[7]) Initially, making any Dataverse API requests from Odyssey resulted in connection errors. This was caused by configuration errors of the internal routing between Odyssey and Dataverse, which were both hosted on Harvard University's servers. This behavior was fixed when Dataverse migrated its API hosting to Amazon Web Services servers from Harvard-internal servers.[8] At the time of writing, this method resulted in the complete download of 810 of 846 unique datasets, where the remaining 36 were unable to be downloaded because DOI-specific metadata was unavailable via the Datavers Native API.

[2] https://github.com/IQSS/dataverse/issues/3597
[3] http://guides.dataverse.org/en/latest/api/search.html
[4] http://guides.dataverse.org/en/latest/api/dataaccess.html
[5] https://github.com/IQSS/dataverse/issues/4529
[6] http://guides.dataverse.org/en/latest/api/native-api.html
[7] https://www.rc.fas.harvard.edu/resources/odyssey-storage/
[8] https://github.com/IQSS/dataverse-client-python/issues/45

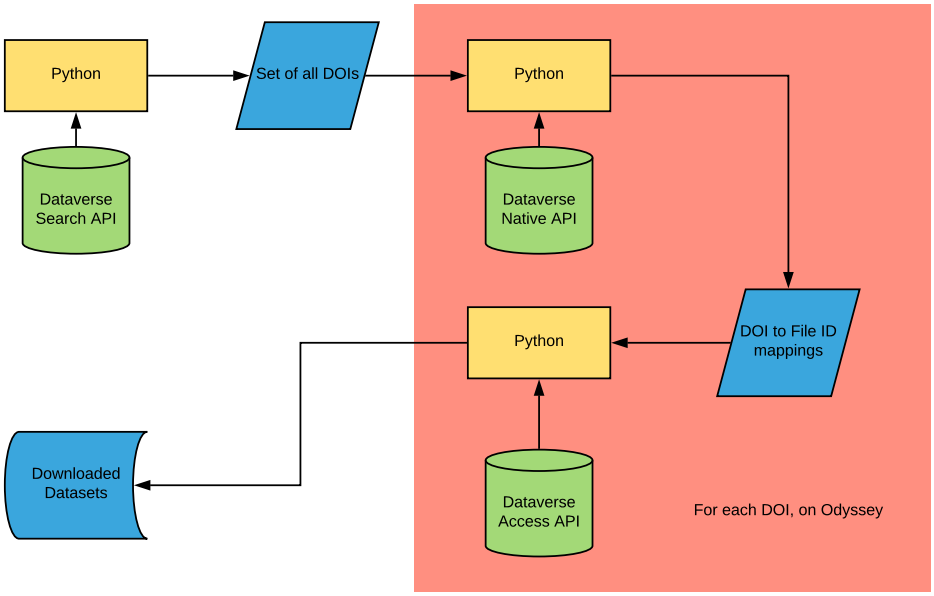

Figure 2.1.1: We queried 3 APIs to download all R-file-containing datasets from Dataverse

2.2 Pre-processing File Encodings

Before running our analyses, we ensured that all downloaded R programs used the same encoding, which refers to the system a computer uses to convert between human-readable characters and sequences of machine-readable binary. Encodings vary across operating systems; MacOS uses the most popular encoding, UTF-8, while Windows allows users to select alternative encodings.[9] Since R can be run on many different operating systems, including Windows, and can also be encoded in multiple formats,[10] two R files containing code which appears the same to humans may have different binary representations. To ensure that differences in encoding would not cause strange errors during execution, we used the Python `codecs` module to convert each R code file from any non-UTF-8 encoding (detected using the Python `chardet` module, which guesses the most likely encoding based on reading files' binary strings) to UTF-8.

2.3 Running Files on Odyssey

With roughly 3,000 R programs downloaded, sequentially recording the execution behavior of each program using a personal computer would have been infeasible. With the datasets already divided into separate directories from the downloading step, we decided to run the analysis for each dataset in its own job on Odyssey.

The first step we took to program this behavior was writing an R script to run every R program within in a specified directory using R's built-in `tryCatch` function (which allows users to control how the program should respond to errors while executing code instead of automatically halting the program execution) in conjunction with the `source` function (which runs in its entirety any R file passed as an argument). Our script recorded "success" when a research program successfully ran, and the first line of error output when a program failed,

[9] https://docs.python.org/3.3/howto/unicode.html

[10] https://support.rstudio.com/hc/en-us/articles/200532197-Character-Encoding

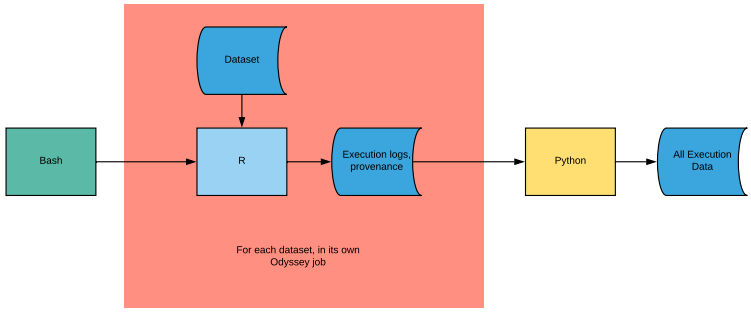

Figure 2.3.1: Workflow for collecting execution for all the R scripts (performed once for original files and once for pre-processed files)

in a file which it placed inside a self-created sub-directory of the dataset (called "prov_data"). After attempting to run each file, the script tried to collect provenance on each file using `ProvR`.

After performing reproducibility analysis for each downloaded dataset, we ran a Python script to correct the most common errors we observed in researchers' R programs. For easy identification of pre-processed files, we added the suffix "__preproc__" to the name of each R script before its ".R" file extension (e.g., "mycode.R" would become "mycode__preproc__.R"; see §3.2 for more details about the pre-processing methodology for specific errors).

We then ran the reproducibility analysis for the pre-processed files. To allow the R script to run for both researchers' original code files and those we pre-processed for common errors, the script accepts a command line argument specifying which type of file (original or pre-processed) to perform and record analyses for.

We deployed reproducibility analysis for every dataset in its own job on Odyssey with 8 cores and a 1-hour run time. Separate jobs were used for analyzing the original and pre-processed versions of the researchers' code. These parameters were chosen to simulate the patience and computational resources available to the typical researcher attempting to replicate another researcher's results. This script structure helped to organize the desired functionality into modular components, and enabled the collection of execution data for nearly 3,000 R files, which would ordinarily take hours if not days to complete sequentially, to finish running in within a few hours.

After all jobs either completed or reached the time limit, we used the `pandas` module in Python to iterate through the dataset directories and aggregate the recorded execution log for each file, enabling us to analyze reproducibility across all datasets.

3 Results

3.1 Error Occurrences by Individual Program

3.1.1 Error Occurrences for Original Source Code

After removing the execution results for datasets which did not finish running within allotted hour, we found the overall success rate of the researchers' R

Execution Result	Count	Percentage (rounded)
Success	408	14.4%
Error	2431	85.6%
Total	2839	

Table 3.1.1: The vast majority of original R scripts failed to execute completely

Error Type	Count	Percentage (rounded)
Library	363	14.9%
Working Directory	696	28.6%
Missing File	802	33.0%
Other	569	23.4%
Total	2431	

Table 3.1.2: 3 errors are responsible for the majority of code execution failures

scripts, which, at 14.4%, was extraordinarily low (see Table 3.1.1)

3.1.2 Common Errors and their Causes

While such a high error rate may have been discouraging from a reproducibility standpoint, it also provided us with a good opportunity to examine the most common error causes. To identify the most common errors, we visually inspected the dataset for error strings which appeared often, and used the Python `pandas` module's built-in filtering options to filter the dataset for their occurrences. Using this method, we were able to identify three types of coding mistakes which accounted for approximately 76.5% of all errors we encountered: library errors, working directory errors, and file errors (see Table 3.1.2).

Library errors involve the `library` function, and usually resemble:

```
Error in library(QMSS) : there is no package called ‘QMSS’
```

This error commonly occurs when attempting to run R code written on a different computer, and indicates that the package the user tried to load (in the example case, `QMSS`) has not yet been installed.

Working directory errors are errors involving the `setwd` function:

```
'Error in setwd(/Users/janedoe/Dropbox/Replication files/) :
cannot change working directory'
```

The `setwd` function[1] changes the directory the R interpreter uses to read, write,

[1] http://stat.ethz.ch/R-manual/R-devel/library/base/html/getwd.html

and search for files. This command makes writing the rest of the script slightly more convenient if all of the script's data dependencies are located in the same directory; the researcher can simply specify files to read in with just their file names and R will look for them in the specified working directory. However, as argued in a blog post by Jennifer Bryan the University of British Columbia,[2] using `setwd` with very specific paths eliminates the possibility of the command executing properly on someone else's computer because different computers almost always have different directory names.

File errors include errors such as:

```
Error in file(file, rt) : cannot open the connection
Error in readChar(con, 5L, useBytes = TRUE) : cannot open ...
Error in read.dta(my_data.dta) :
    unable to open file: 'No such file or directory'
```

Though these error messages are different, they are all caused by trying to import data from an invalid file name or location.

The remaining errors, classified under "Other", were somewhat more difficult to identify, and usually involved specific syntax mistakes in the R source code files. However, the fact that only three types of errors account for approximately 76.5% of all the errors we observed suggests that reproducibility of R-based research can be dramatically improved if researchers make just slight changes to their coding habits. For example, the `setwd` function can often be omitted entirely from the R script. Instead, if data files are located in the same directories as analysis scripts, researchers just need to call `setwd` once for the relevant directory from the R console prior to running their analyses. Even if we could fix all occurrences of these three mistakes, we most likely cannot fix all 76.5% of errors because these basic errors could be masking more pernicious ones. To investigate this further, we attempted to fix these basic errors automatically (see §3.2) and analyzed the execution results after doing so (see §3.1.3).

[2] https://www.tidyverse.org/articles/2017/12/workflow-vs-script/

Error Type	Count	Percentage (rounded)
Library	8	0.3%
Working Directory	12	0.5%
Missing File	1400	60.1%
Success	62	2.6%
Other	847	36.4%
Total	2329	

Table 3.1.3: Most of the programs originally containing library and working directory errors also contained other errors, with the vast majority attempting to import data from missing files

3.1.3 Error Occurrences for Pre-Processed Source Code

After applying our procedure for automatically correcting library, working directory, and file errors, we ran the reproducibility analysis for all pre-processed scripts. Because there was some variability in the number of files which ran to completion within the 1-hour time limit, especially if pre-processing a program allowed it to execute longer than its corresponding original (which may have failed nearly instantly), we only have data on 2,329 of the pre-processed counterparts to the 2,431 original files which produced errors (See Table 3.1.3).

Our pre-processing methods fixed 62 of the 2,329 files which originally contained errors. Though we were initially disappointed by this result, we found that it was due mostly to the egregiously large number of missing files and the masking of other errors by the three originally-most-common types.

Our pre-processing reduced the total number of library and working directory errors from over 1,000 to 20. All of the remaining library and working directory errors took these respective forms:

```
Error in library(rstan, lib.loc = ~/R/win-library/3.2) :
    no library trees found in 'lib.loc'
Error in setwd(paste(mywd, /Data, sep = )) :
    cannot change working directory
```

Because these were complex cases which occurred very infrequently, we did not think these results were a cause for concern.[3]

The most striking result was the large increase in the number of missing file errors, from 696 to 1,400. To confirm that these were due to missing files, we added some functionality in our Python file error correction function to log whenever it failed to find a path for a file loading command while parsing an R program. With our fairly naive string parsing and file searching approach (see §3.2.3), we confirmed that above 86% of the scripts which produced file errors referenced files which were simply missing from the dataset (and probably failed to confirm the remaining 14% because our list of data-importing functions was not completely exhaustive). This result emphasizes that even syntactically correct code will not generate reproducible results in the absence of the necessary dependencies. We suspect that researchers often simply forgot to upload all these dependencies, which would be quite easy to do if their scripts have many data inputs. Provenance-aware tools which detect these errors and report them back to the researcher *before* they publish their datasets would be extremely useful in these cases. For instance, such a feature could be integrated into the data upload form for Dataverse to prevent these errors from occurring in the future.

3.1.4 Failure Rates over Time

To examine how R's exploding popularity over time relates the reproducibility of R-based research, we used Python's `requests` module to query the Dataverse Native API for release time of the dataset's latest version, then Python's `pandas` module to extract the year of release and combine the release time and execution data for simultaneous analysis. After summing up the total file count and error count for each year to calculate the error rate, we observe a surprising result: though the number of R files published to Dataverse has demonstrated steady

[3] It was difficult to use Python string processing to capture all the parameters of these function calls. Because our error correction procedure is meant to be used as a last resort and only corrects a handful of errors in the first place, we chose not correct these corner cases for the purposes of this analysis.

Year	Total Files	Total Error Files	Error Rate (Rounded)
2015	282	253	89.7%
2016	927	786	84.8%
2017	1416	1204	85.0%
2018	212	186	87.8%

Table 3.1.4: The yearly error rate of R files published to Dataverse has remained stable despite large increases in yearly R files published

growth, the proportion of files which fail to run has stayed remarkably stable between 85-90% (See Table 3.1.4). The small numbers for 2018 are likely due to the timing of our analysis of writing in March, 2018.

Visualizing this data with Python's `seaborn` module, we can observe just how well the proportion of files which error has kept pace with the increases in R files published. This finding shows that while the popularity of R-based research in the Harvard Dataverse has increased substantially, the reproducibility of said research has remained consistently poor (see Figure 3.1.1). This finding lends particular urgency to improving the reproducibility of R-based research, because it implies that the body of difficult-to-reproduce research is growing at an increasing rate. It also demonstrates that the errors we found in the R programs were more-likely the result of coding mistakes than code deprecation; if the errors were the result of aging code, we would expect the error rates of the R programs to decline for code published more recently, which was not the case. Thus, the R programs did not decline in reproducibility as they aged. They were never reproducible in the first place.

3.1.5 Failure Rates by Subject

The Dataverse Native API also provided metadata for the subject of the dataset's corresponding article. Dataverse was originally focused on sharing data from the social sciences,[4] the R-program-containing datasets on Dataverse cover a

[4] https://dataverse.org/about

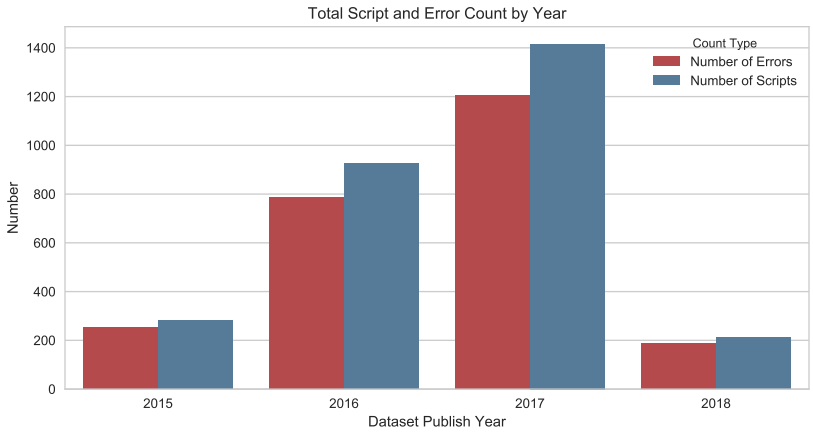

Figure 3.1.1: Increases in the yearly quantity of R code published were accompanied by increases in the number of files which fail to execute

multitude of additional subjects, including: Computer and Information Science, Medicine, Health and Life Sciences, Physics, Engineering, Business and Management, Mathematical Sciences, Arts and Humanities, Agricultural Sciences, Law, and Earth and Environmental Sciences. Even though the dataset-to-subject relationship is one-to-many, since each dataset can cover multiple subjects, we still thought the relationship was worth investigating to see if certain subjects had substantially lower error rates. Using the Python `pandas` module, we encoded binary variables for each subject and combined the data with our execution data. We then calculated the within-subject script count and failure rate (see Table 3.1.5, Figure 3.1.2)

In alignment with expectations, the Social Sciences label was the most popular, with over 2,600 associated R scripts and an error rate very representative of the overall error rate at 88.3%. The amazingly low error rate for Engineering is likely because the of the small sample size of Engineering-labelled R scripts, as shown by its large error bar and standard error of 17.9%. On the other extreme, the 100.0% error rates of Physics and Business and Management may appear

Subject	Error Rate (Rounded)	Std. Error (Rounded)	Program Count
Social Sciences	87.2%	0.6%	2677
Computer and Information Science	95.2%	3.3%	42
Medicine, Health and Life Sciences	54.7%	4.8%	106
Physics	100.0%	0.0%	22
Engineering	20.0%	17.9%	5
Other	60.7%	9.2%	28
Business and Management	100.0%	0.0%	14
Mathematical Sciences	42.6%	6.0%	68
Arts and Humanities	90.0%	6.7%	20
Agricultural Sciences	81.3%	9.8%	16
Law	77.2%	5.6%	57
Earth and Environmental Sciences	54.6%	4.8%	108

Table 3.1.5: Different subjects had dramatically different error rates

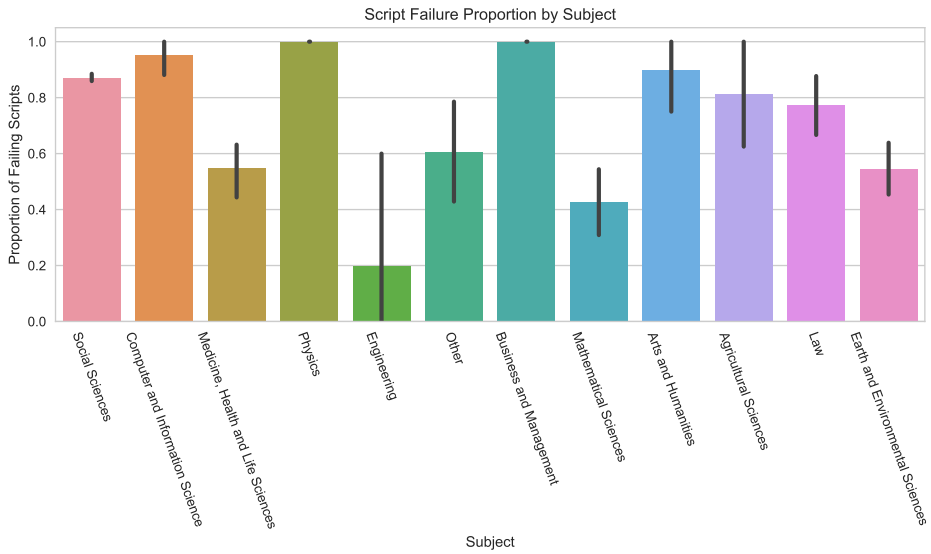

Figure 3.1.2: There are striking differences in error rate by subject, but because Social Sciences are disproportionately represented on Dataverse, this may not be a generalizable effect

alarming, especially coupled their 0% standard errors and lack of error bars. However, we do not think the results for these two subjects are significant enough to interpret because both also had a relatively low number of associated R programs (22 and 14, respectively), and their 0% standard error values and lack of error bars were simply a property of the formula for calculating the standard error of binomial proportions.[5] We thought the most interesting trends were the significantly lower error rates of scripts labeled with Mathematical Sciences; Medicine, Health and Life Sciences; and Earth and Environmental Sciences, with respective error rates of 42.6%, 54.7%, and 54.6% and relatively narrow error bars. We cannot be certain whether different coding and data integrity practices exist within these subjects or whether these findings were purely incidental, however, we think this effect is significant enough to motivate further investigation.

3.1.6 Robustness of ProvR

We found that executing the file with `ProvR` succeeded in 92.6% of the cases where the file successfully executed with just the `source` call (375 of 405 files). Because `ProvR` had never been tested with such a large volume of code before, its performance can be considered decent. However, a tool designed to facilitate reproducibility should never introduce errors, suggesting that more work is still needed to make `ProvR` more robust.

3.2 Pre-processing to Correct Common Errors

3.2.1 Library Errors

To automatically correct library errors, we used string processing in Python to replace all calls to package-loading the package loading functions, `library` and

[5] a proportion calculated from a collection of independent random binary outcomes (Bernoulli-distributed), where each member of the collection has the same assumed probability of either outcome; the standard error of the estimated proportion will be 0 if one of the two two possible binary values (in our case, these values were script execution success/failure) was never observed

```r
if (!require("stringr", character.only=TRUE)){
    install.packages(pkgs="stringr", repos="http://cran.r-project.org")
    require("stringr", character.only=TRUE)
}
install_and_load <- function(x, ...){
  # if the input is a string
  if (is.character(x) & length(x) == 1) {
    # check if there are commas in the string
    if (grepl(",", x)) {
      # change x to a vector of strings if there are commas
      x = str_split(x, ",")[[1]]
    }
  }
  for (package in x) {
    if (!require(package, character.only=TRUE)){
      install.packages(pkgs=package, repos="http://cran.r-project.org")
      require(package, character.only=TRUE)
    }
  }
}
```

Figure 3.2.1: A more robust function for loading packages.

`require` (which yields a warning instead of an error like `library` if package loading fails), and the package installing function, `install.packages` with a function we wrote, `install_and_load`, loosely based on a series of Stack Overflow answers (see Figure 3.2.1).[6]

In order to improve the likelihood that their code will run on other computers, some researchers call `install.packages` on every package they use in their R scripts. This method is effective but inefficient, as R will re-install already-present packages. In contrast, `install_and_load` only installs packages if they are not already installed. To avoid introducing new errors with pre-processing (which would be definitionally counter-productive towards our goal of improving reproducibility) Python script inserts the function definition for `install_and_load` at the beginning of the R file and anytime the script calls the `rm` function, which clears the user's functions and variables from memory. While not the most elegant solution, this approach ensured that `install_and_load` would always defined when called.

[6] https://stackoverflow.com/questions/9341635/check-for-installed-packages-before-running-install-packages

3.2.2 Working Directory Errors

We attempted to correct incorrect paths passed to the `setwd` command using Python string processing combined with the `os` module. Our pre-processing function first searches the researchers' datasets for any directories resembling the path they specified, correcting the path if one is found. For example, for the command `setwd("path/specific/to/own/computer")`, our function first searches for the inner-most, "computer", directory, and if that directory does exist, searches for "own/computer", then "to/own/computer" and so on until the search path matches the entire original path. If none of these searches succeed, the function performs a search of the entire dataset's directory structure (including any nested directories) for the most specific path ("computer" in our example), before omitting the command entirely from the source code file — a solution which often works if the dataset contains no nested directories and the data and source code therefore live in the same directory.

3.2.3 File Errors

File errors were the most difficult to correct because *any* R function for importing data could potentially cause a file error. Over-generalizing and attempting to correct all strings in the R file would be inefficient because of the many uses of strings in R, and prone to false positives, for example, a string passed to a function as a parameter could be interpreted as a potential file name, and attempting to fix it could introduce new errors.

Our approach, therefore, was to build a list of the most common sub-strings that appear in the names of R data importing functions based on a comprehensive web-based guide to importing data in R. [7] These sub-strings include: `read`, `load`, `fromJSON`, `import`, and `scan`. We used Python string processing to search each line of code for these sub-strings and the `re` regular expression module to capture the first string parameter or the `file` argument of

[7] https://www.datacamp.com/community/tutorials/r-data-import-tutorial

any subsequent function calls if the regular expression returned a match. We then used the search method described in above §3.2.2 with the minor change that the target of the search was a file instead of a directory. In keeping with our conservative philosophy that we should avoid introducing new errors at all costs, the Python function makes no modifications to a line of code if no file matching the original file name was found.

We did not attempt to correct improper output file paths. Even though the output file itself need not be present for an output path to be correct (since the output function would just create the file), incorrect intermediate directories in the output file would certainly cause an error. For example, "output.csv" would always be a valid output path, while "unreproducible/code/directory/output.csv" would only be valid if all the intermediate directories were present. A method similar to the one we used to detect importing functions could be used to detect output functions and amend (or delete) any intermediate directories, but we considered the risk of introducing new errors too high to implement a more proactive path replacement policy, and changing the output directory may break the script in other, harder-to-detect ways (for example, if the code later attempts to import data from a file outputted earlier in the same script).

4

containR

4.1 Original Implementation Plan

Packages like `encapsulator` [4] which collect and use provenance data have not seen widespread adoption despite their indisputable usefulness, likely because of the non-trivial effort required to install them and learn their associated sytaxes. To ameliorate both of these inconveniences, we sought to produce a web-based application with a user friendly graphical interface with `encapsulator`-like functionality that includes the error pre-processing methods we developed for assessing reproducibility.

We originally thought the most straightforward way to achieve `encapsulator`-like functionality was to use `encapsulator` itself — why reinvent the wheel? `encapsulator` offers powerful functionality, including built-in provenance collection, code-cleaning options, the automatic

downloading and installation of code dependencies based on captured provenance, and automatic generation of a Linux virtual machine with the code inside, runnable on nearly all personal computers with the addition of popular hyper-visor software. However, upon installing `encapsulator` using the instructions provided in the paper [4], this software designed to improve code reproducibility became unresponsive while running *its own* example code.[1] Additionally, after getting `encapsulator` running, the process of assembling a virtual machine with the necessary dependencies using Vagrant [2] took several minutes and required the virtual machine itself to be opened in the process. While virtual machines offer both power and security, allowing a web server to open at least one for each concurrent user seemed a recipe for latency. Additionally, `encapsulator` was built using Ruby, a language with which the author has no experience and which would have caused awkward interfacing with Python web framework.

With these concerns in mind, we implemented a web application which provides similar functionality to `encapsulator` natively in Python, save for the provenance collection component, for which we used `ProvR` [5]. We also replaced automatic virtual machine generation with the automatic generation of a Docker image. Docker "containers" offer virtual-machine-like functionality for most applications at a fraction of the resource cost. The Docker website explains :

> A container runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.
>
> By contrast, a virtual machine (VM) runs a full-blown "guest" operating system with virtual access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.[3]

[1] https://github.com/ProvTools/encapsulator/issues/23
[2] https://www.vagrantup.com/intro/index.html
[3] https://docs.docker.com/get-started/#images-and-containers

Docker images are immutable files containing the file dependencies needed for producing a container, the actual executable that runs the contents of an image. Our web application's core functionality is the automatic generation of Docker images that bundle researchers' code and data, RStudio server, [4] all package dependencies, and provenance data. These images are generated based on Dockerfiles, text files specifying how the Docker image should be built which we modify to include instructions to download and install the required pacakges. Docker images take seconds, as opposed to several minutes (as was the case with virtual machines) to generate, and are easily uploadable to the Docker Hub, a platform for storing and sharing Docker images analogous to Github for code repositories, [5] for easy sharing (see Figure 4.2.6).

Switching to Docker provided some additional benefits as well, including the existence of open-source, curated Docker images specifically built to run RStudio Server with the core R packages pre-installed, [6] and the Docker Native Python client [7] which interfaced elegantly with the Python back-end of the website. With the prominent role played by Docker containers in our application's workflow, it was only fitting to name the application `containR`.

4.2 Functionality Overview

4.2.1 Registration and Login

Example

After registering and logging in (see Figures 4.2.1, 4.2.2), users are taken to their home page, which lists all the Docker images they have created in chronological order. The "Docker Images" section of the home page is empty when our

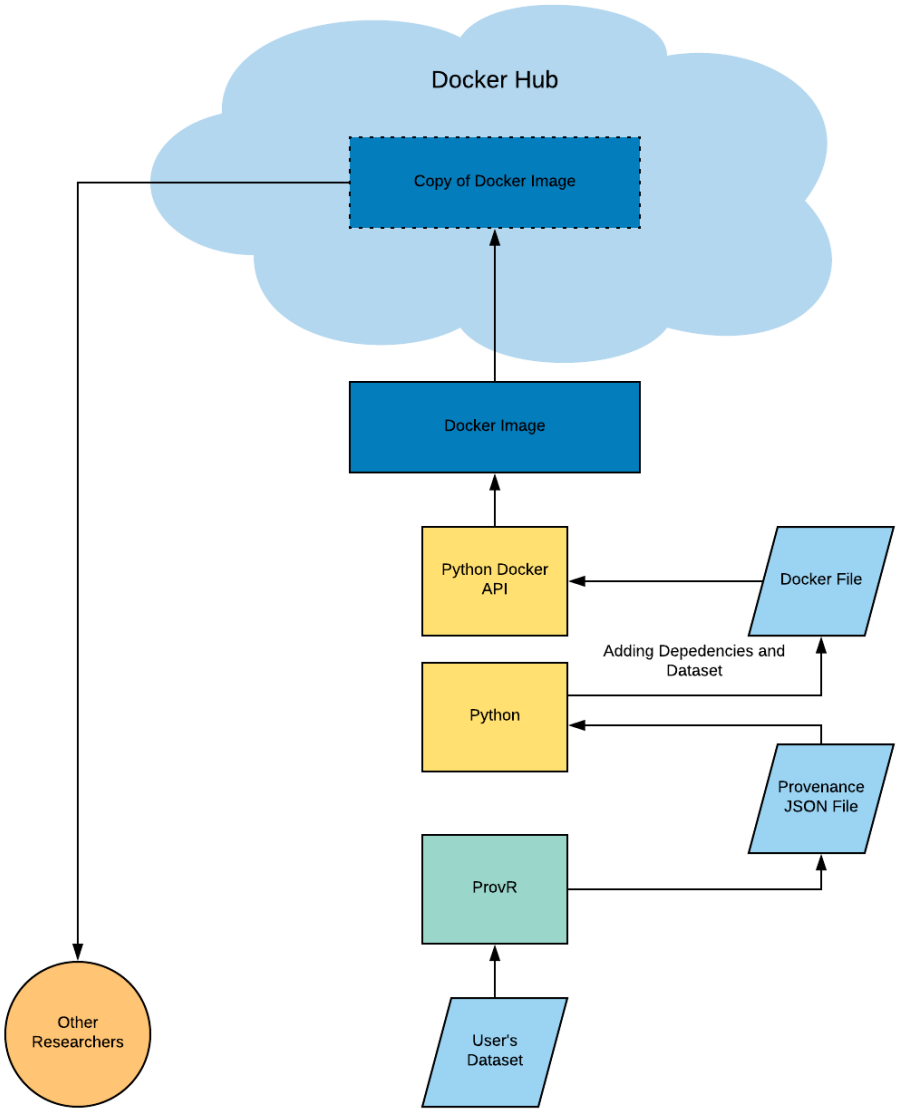

Figure 4.1.1: `containR`'s process for generating a Docker image based on a user-provided dataset

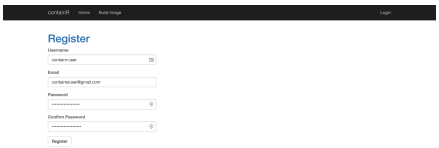

Figure 4.2.1: containR Registration Page

Figure 4.2.2: containR Login Page

Figure 4.2.3: containR Home Page

example user signs in for the first time (see Figure 4.2.3).

Implementation Details

The boilerplate code, including the user database, user registration and login management, and basic front-end were implemented using code from Miguel Grinberg's Flask tutorial. [8] To manage each user's personal data and created

[4] https://www.rstudio.com/products/rstudio/
[5] https://hub.docker.com/
[6] https://github.com/rocker-org/rocker
[7] https://docker-py.readthedocs.io/en/stable/
[8] https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

Docker images, containR interfaces with a sqlite[9] database using Flask-SQLAlchemy, [10] which allowed us to create and manage database relationships using Python objects instead of raw SQL, thereby significantly increasing the interpretability and cohesiveness of the back-end code base.

4.2.2 Loading Data and Selecting Pre-Processing Options

Example

To interact with containR's core functionality, logged-in users select the "Build Image" option on the website's navigation bar, which takes them to a form for loading a dataset into containR (see Figure 4.2.4). The form provides users with two options for loading in the dataset: (1) providing the Harvard Dataverse DOI, or (2) providing a .zip file containing the data and code. Here, we demonstrate the "DOI" option by entering in "doi:10.7910/DVN/U1GGGQ", the unique identifier of a dataset hosted on Harvard's Dataverse from Banda et al. which explores how negative advertising affects citizens' perceptions of political candidates [12].[11] We also select the options for automatically cleaning the code and fixing code errors (see §3.2). After we click the "Build Docker Image" button, containR will download the dataset based on the DOI we entered, perform the pre-processing we requested, and update our home page with a link to our newly-created data archive (Docker Image).

Implementation Details

The dataset upload form was built using the Python Flask extension, Flask-WTF,[12] which provides streamlined tools both for generating form fields and verifying form responses. We used Flask-WTF to create all of the fields in

[9] https://www.sqlite.org/index.html
[10] http://flask-sqlalchemy.pocoo.org/2.3/
[11] https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/U1GGGQ
[12] https://flask-wtf.readthedocs.io/en/stable/

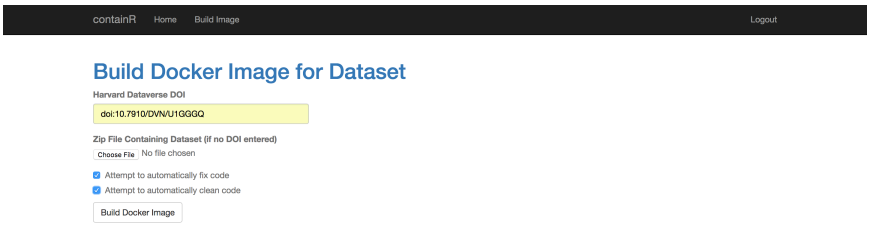

Figure 4.2.4: Build `containR` form, with options for correcting and cleaning the code

our form, including the file-upload field. After receiving the form data from the user and performing form validation using `Flask-WTF`, `containR` undergoes a five-step process:

1. Obtain the dataset either from the user-uploaded .zip file or by downloading it from Dataverse through the Data Access API
2. Perform user-requested pre-processing, including error correction and cleaning [13]
3. Collect provenance data on the code using `ProvR`
4. Parse the provenance to create a Dockerfile specifying the R package dependencies to install and the location of the user's dataset
5. Create a Docker image and upload the image to Docker Hub using the Python Docker Client

[13] At the time of writing, the code correction functionality had been incorporated into the website, but `Rclean` was still being actively developed, so the code cleaning field was just a placeholder

This process may take several minutes to complete because ProvR collects provenance on R programs while they are executing. Thus, if the R programs have long execution times, collecting provenance will take just as long. To avoid freezing the front-end of the website during this potentially-lengthy process, all of the code analysis and Docker-image building are performed in a new thread created using Python's `threading` module. We intend to migrate this functionality to use the `celery` [14] task queue system instead, which provides a lightweight and efficient way to run tasks in the background while communicating status updates to the front-end, which would allow us to provide containR users with intermediate status updates during the Docker-image build process.

4.2.3 Docker Hub and Running the Docker Image

Example

After containR finishes generating and uploading the Docker image containing our dataset, our home page will be updated with a link to the dataset on Docker Hub (see Figure 4.2.5).

The URL for the newly-created entry on the home page takes us to the Docker image's page on Docker Hub. Our image's Docker Hub page (see Figure 4.2.6) shows basic information about our image as well as the command to pull the image onto a personal computer.

The images containR creates on Docker Hub are completely public, allowing total transparency and accessibility for our example analysis of negative political advertising. The name for our example image is "containr/containr-test-uptxs", and was randomly generated during the image building process. Our political advertising dataset, including all the code, code dependencies, and data, are now available for download by anyone with Docker installed.

[14] http://flask.pocoo.org/docs/0.12/patterns/celery/

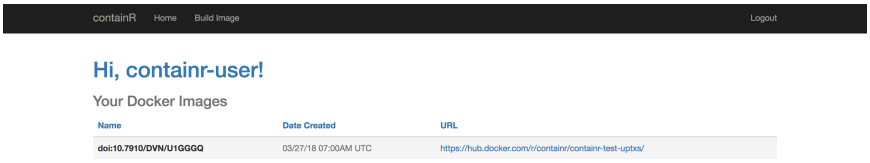

Figure 4.2.5: After `containR` finishes building a Docker image, the user's homepage is updated with a link to the image on Docker Hub

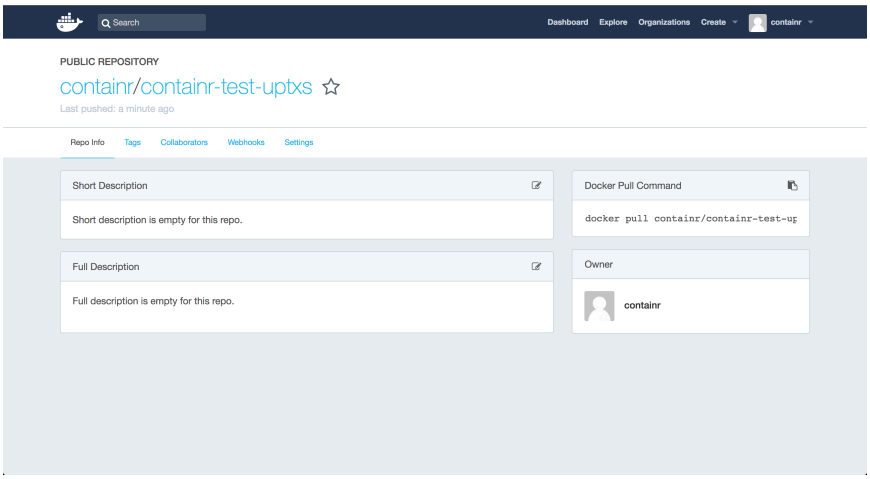

Figure 4.2.6: The user's image on Docker Hub can be shared with anyone

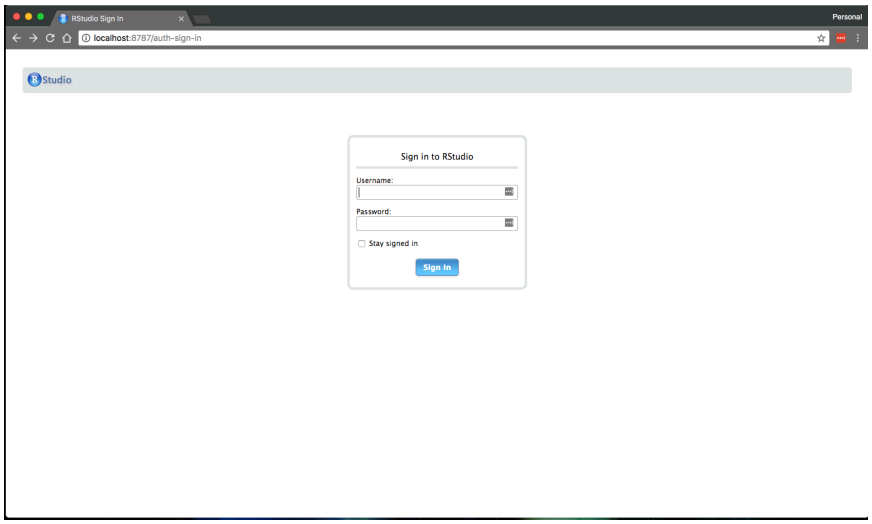

Figure 4.2.7: Users log in to their local RStudio Server with username and password "rstudio"

If we wanted to retrieve the data at a later time or if we were playing the role of a researcher trying to verify the results, we can download the Docker image by running the following command in our terminal:

```
docker pull containr/containr-test-uptxs
```

After the image download has completed, we directly launch a container from the image we just downloaded with the following command:

```
docker run -p 8787:8787 containr/containr-test-uptxs
```

Within seconds of executing `docker run`, we can visit `localhost:8787` in our preferred web-browser, which brings us to the RStudio Server login page. After we log in with the username and password "rstudio" (see Figure 4.2.7), we are immediately greeted with the familiar RStudio user interface. The dataset from our political advertising data appearing in the files pane in the bottom-right of the screen (see Figure 4.2.8).

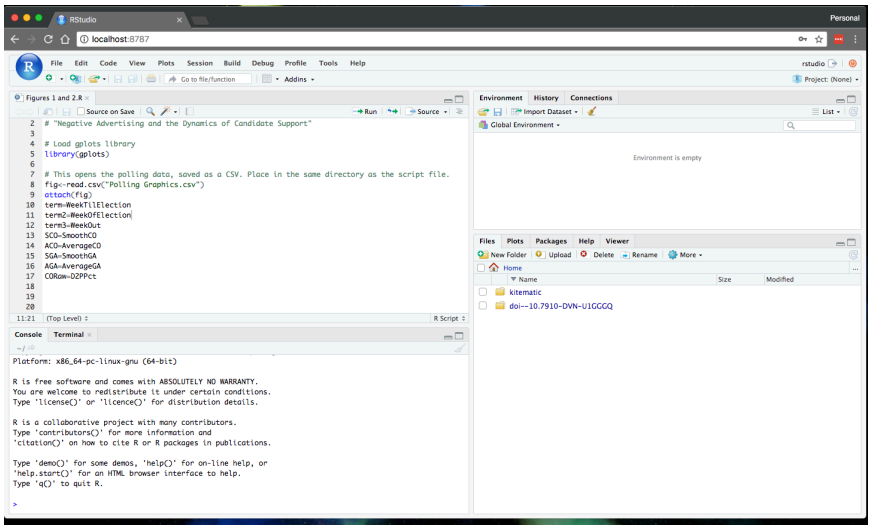

Figure 4.2.8: The RStudio Server instance comes pre-loaded in the image with the researcher's code, data, and package dependencies

If we look closely at the code in Figure 4.2.8, we observe that the code calls the `library` function to load the R package dependency `gplots`. If we had just downloaded the code and data for Banda et al.'s experiment from Dataverse, we would likely have to install this library ourselves (assuming we had never used it before) in order for the code to execute correctly. However, since `containR` pre-loaded all the package dependencies while generating the Docker image, we can immediately run all the code contained in Banda et al.'s dataset (after ensuring that we are in the correct working directory), and regenerate the results the dataset was designed to reproduce (see Figure 4.2.9).

By taking about a minute to fill out the "Build Image" form for `containR` and turning our dataset into a Docker image, we have guaranteed the future reproducibility of Banda et al.'s work and made their work far more accessible for researchers to reproduce in the future; a researcher interested in Banda et al.'s analysis can re-run and extend the analysis within minutes with only Docker and their favorite web-browser.

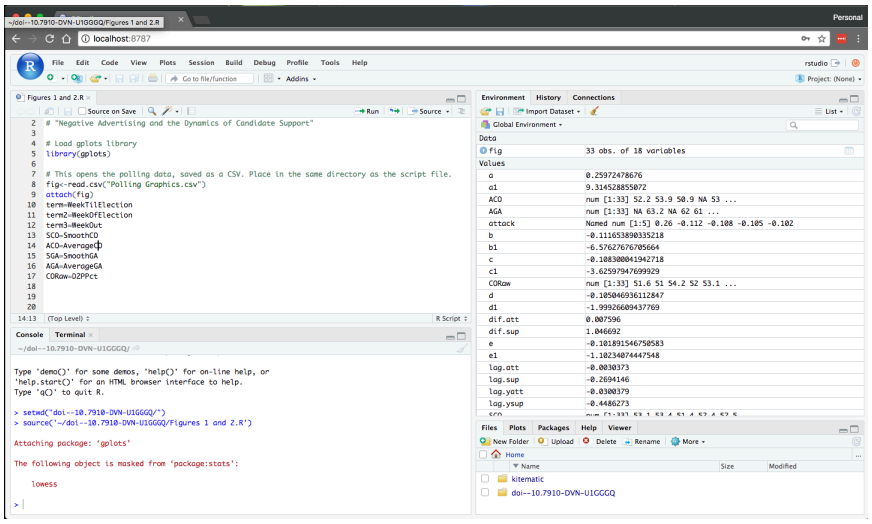

Figure 4.2.9: Once the Docker image is successfully built, the research code inside can be run on any computer with Docker and a web browser installed

Implementation

Most of the functionality described in the example above are attributable either to Docker (image creation, publishing to Docker Hub, and running an image as a container) or Rocker[15] (pre-installing RStudio Server inside Docker images). However, we would like to note that multiple Docker containers can be run simultaneously on a user's computer with different port numbers.[16] Thus, researchers can simultaneously interact with multiple datasets from different sources and with completely different code and data dependencies using different images pulled from Docker Hub. We believe this ability to execute multiple researchers' R programs at once will also facilitate the process of reproducibility.

[15] https://github.com/rocker-org/rocker

[16] https://github.com/rocker-org/rocker/issues/280

5

Conclusion

With our analysis of nearly 3,000 R scripts from published datasets in Harvard University's Dataverse, we found that between 85% and 90% of the R code published each year does not execute on the first attempt. While researchers attempting to reproduce results may be willing to correct these errors themselves, the time it costs them to do so would certainly be more-productively spent advancing their respective fields. Our finding that the rapid increase in R code published to Harvard's Dataverse each year since 2015 has left the yearly error rates unaffected suggests that the problem is with the R programs themselves and not the age-influenced breakage of code dependencies. The overwhelming prevalence of errors caused by missing data files in the datasets we analyzed suggests that better care must also be taken on the producing researcher's side to be rigorous and diligent when publishing their datasets.

The additional effort required for researchers to improve their research's

reproducibility can be reduced with tools like `containR`, a web application which checks their code for executability and guarantees future executability with the use of Docker images. With its web-based user interface, minimal dependency requirements (only Docker and a web browser), and automatic code correction and cleaning, `containR` is likely the most straightforward data-provenance-leveraging tool yet built, but convincing people to change their workflow always requires incentives. To provide these incentives, journals and online data repositories like Dataverse could require, for example, that a link to an image on Docker Hub be included with article submissions. Our finding that R code from datasets in certain subjects (like Earth and Environmental Sciences and Mathematical Sciences) were more likely to execute completely than code taken from others, suggesting that it may be beneficial for researchers in different disciplines to educate each other on a set of best coding practices in R and hold each other accountable for following it.

By expounding on the poor reproducibility of the large growing volume of R code in the Harvard Dataverse, we do not mean to imply ill will towards the dataset-producing researchers or comment on the quality of the undoubtedly innovative research they conduct. Rather, our finding that over 70% of errors in the R code we examined can be attributed to just 3 common coding errors shows that with slightly more mindfulness, researchers can dramatically improve the accessibility of their code and accelerate scientific progress for all. The code we used for our reproducibility analysis[1] is publicly available on Github, and the code for `containR` will soon follow.

5.1 Future Work

While our analysis of reproducibility was limited to the R code in the Harvard installation of Dataverse, our pipeline for obtaining data using the Dataverse API could be leveraged for performing similar reproducibility analyses for other coding languages and Dataverse installations.

[1] https://github.com/cscn/thesis

Another extension on our research could involve finding more applications for provenance data. As a consequence of running `ProvR` on many of the datasets we downloaded, we now have a large collection of provenance JSON files, labeled by DOI, along with metadata for the code that generated the data and a pipeline for accessing more metadata by querying the Dataverse Native API. We are excited to see if and how this data provenance dataset can be used to advance provenance-aware research.

As part of the error analysis and build process for `containR`, we produced a small library of helper functions for pre-processing R files for common errors. With a little bit of work, these functions could be implemented natively in R for real-time code correction. Alternatively, the R package `reticulate`[2] may be promising for this purpose because it allows users to interact with Python objects from within a running R session.

At the time of writing, `containR` was not yet ready for production deployment. Our goal is to have `containR` hosted on Amazon Web Services so researchers can begin experimenting with it within the next few months. Some additional functionality which needs to be implemented is securing the execution of user-provided code to prevent malicious or badly written code from damaging the web server, reporting code execution errors back to users, and support for code cleaning using `Rclean`.

[2] https://cran.r-project.org/web/packages/reticulate/index.html

References

[1] V. Stodden, J. Seiler, and Z. Ma, "An empirical analysis of journal policy effectiveness for computational reproducibility," *Proceedings of the National Academy of Sciences*, vol. 115, no. 11, pp. 2584–2589, 2018.

[2] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," *Proceedings of the 2017 Symposium on Cloud Computing - SoCC 17*, 2017.

[3] B. S. Lerner and E. R. Boose, "Rdatatracker and ddg explorer," *Lecture Notes in Computer Science Provenance and Annotation of Data and Processes*, p. 288–290, 2015.

[4] T. Pasquier, M. K. Lau, X. Han, E. Fong, B. S. Lerner, E. Boose, M. Crosas, A. Ellison, and M. Seltzer, "Sharing and Preserving Computational Analyses for Posterity with encapsulator," *ArXiv e-prints*, Mar. 2018.

[5] "Provtools/provr." https://github.com/ProvTools/provR.

[6] S. Bowers, T. Mcphillips, B. Ludäscher, S. Cohen, and S. B. Davidson, "A model for user-oriented data provenance in pipelined scientific workflows," *Provenance and Annotation of Data Lecture Notes in Computer Science*, p. 133–147, 2006.

[7] P. Missier, K. Belhajjame, J. Zhao, M. Roos, and C. Goble, "Data lineage model for taverna workflows with lightweight annotation requirements," *Lecture Notes in Computer Science Provenance and Annotation of Data and Processes*, p. 17–30, 2008.

[8] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva, "Tackling the provenance challenge one layer at a time," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, p. 473–483, 2008.

[9] "The r project for statistical computing." https://www.r-project.org/.

[10] "Rclean." https://cran.r-project.org/web/packages/Rclean/index.html.

[11] "Dataverse." https://dataverse.org.

[12] K. K. Banda and J. H. Windett, "Negative advertising and the dynamics of candidate support," *Political Behavior*, vol. 38, p. 747–766, Jul 2016.

Colophon

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. A template, which can be used to format a PhD thesis with this look and feel, has been released under the permissive MIT (X11) license, and can be found online at github.com/suchow/ or from the author at suchow@post.harvard.edu.