Abstract

The use of computers for data processing and analysis has dramatically transformed the approaches and capabilities of scientific research. Today, researchers are able to process and draw conclusions from large volumes of data in relatively little time, expanding the breadth and efficiency of their work. Despite this shift, verifying results through multiple studies and experiments will always remain important. A 2019 National Academies report recommended more research and development to ensure published scientific results are computationally reproducible, meaning the same results can be derived from the original data and analysis methods.[1] Often, computational reproducibility requires information about the computing environment – such as the operating system, language, and package versions where the results were produced – as well as the data and script. This is because software can behave differently when components of the computing environment change. Therefore, an approach to reproducible research involves collecting all of the information about the scripts, data, and computing environment, also known as data provenance. In the R language, the rdtLite package facilitates the collection of data provenance for a given script execution. This thesis will focus on developing methods that use data provenance as a blueprint for reconstructing a computing environment and conducting experiments that apply this tool to identify situations in which changes to the environment resulted in changes in script behavior.

[1] National Academies of Sciences, Engineering, and Medicine. 2019. Reproducibility and Replicability in Science. Washington, DC: The National Academies Press. https://doi.org/10.17226/25303.

MOUNT HOLYOKE COLLEGE SENIOR THESIS

Using Data Provenance to Support Reproducibility in R

by Sean Nicole Fabrega

Thesis Advisor: Professor Barbara Lerner

Presented to the faculty of Mount Holyoke College in partial fulfillment of the requirements for the degree of Bachelor of Arts with Honors

May 2023

Acknowledgements

I would like to thank the faculty and staff of the Mount Holyoke College Computer Science Department for cultivating my passion for this subject and with each class, tediously unpacking difficult concepts, answering every question, and supporting me through all the excitement, confusion, and frustration that comes with this field. I want to specifically acknowledge Professor Melody Su, Professor James "Murphy" McCauley, Charles Romer, and Barbara Dalton Rotundo for their support and guidance.

To everyone at Harvard Forest, including my mentors, student cohort, as well as Nautica Jones and Savanna Brown, thank you for supporting me during the summer when I started researching and developing tools for data provenance use. Thank you to the National Science Foundation for funding my summer research.

Thank you Willow for helping me collect scripts for my research.

To my thesis advisor, Professor Barbara Lerner, thank you so much for guiding me through this research process. Thank you for your support, expertise, and patience. I am so glad I had the opportunity to work on this project with you.

I want to thank my two best friends, my moon and my sun, Meredith and Claire, for supporting me, bringing me treats, and reminding me to take breaks.

I also want to thank Thomas for his encouragement, love, and proofreading expertise.

Lastly, I would like to thank my family. Thank you Kim, for your wisdom and your delicious food. Thank you Sydney, for putting up with my shenanigans and somehow knowing me so well. Thank you Dad. Your drive and dedication to your odd computer science niche inspire me. Thank you Momma, for always reminding me to not be so hard on myself.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 Motivation

Data is almost exclusively processed and analyzed using the power of computers, and thus the development and maintenance of computer software have revolutionized how we observe and understand our world. However, verifying results and conclusions from data analysis remains an imperative part of the research process. Without additional replication and reproduction of scientific results, the accuracy or truthfulness of the result is unknown. When scientists observe consistency in the behavior of studied phenomena through the approaches of replication and reproduction, the conclusions made about the world become more convincing. Therefore, these processes continue to be of the utmost importance for researchers.

In 2019, the National Academies of Science, Engineering, and Medicine published a report titled "Replicability and Reproducibility in Science" to address these issues, especially in the context of waning public trust in science [1]. They defined replicability as "obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data" and reproducibility as "obtaining consistent computational results using the same input data, computational steps, methods, code, and conditions of analysis" [1, p. 1]. In other words,

replicability involves using similar scientific methods to collect new data, ultimately attempting the production of consistent results. In contrast, reproducibility describes using the same specific methods of analysis, in most cases the code used, on the same dataset to prove the consistency of the result. They emphasize that their definition of reproducibility stems from the growing issue of computational reproducibility in scientific research, citing the rise in the use of computers and large databases for research in all disciplines, not just those which are computationally focused. The report stresses a couple of issues with this trend, most of which involve a lack of transparency. Often, data analysis processes are not public or accessibly archived, which can be problematic because minute differences in the script or computational environment can drastically affect results. Without the original scripts and data, "computational reproducibility" is not possible by definition. While some new tools capture information about changes to the data analysis process tasked to address this issue, such as various software packages which collect data about the computational environment during the execution of a script, they are not widely used by scientists. Furthermore, there is no universally consistent methodology for archiving information regarding different computational research processes [1, p. 7]. If other researchers cannot access these digital artifacts, computational reproducibility becomes virtually impossible. Therefore, some scientists advocate for the required release of relevant code and data alongside scientific results, sometimes called "open science," stating that anything less severely restricts reproducibility in computationally reliant parts of research [4][20].

Even when these processes are properly archived and accessible to researchers, the original software and data used during the research process may no longer produce the same result over time because of the variability of the computing environment, such as different software and hardware components. For example, the now-common practice of using floating-point numbers in computational data analysis creates potential discrepancies in the value of results because different hardware

and software treat their calculations differently [25]. Furthermore, alternative versions of the same software may produce distinctly different results. Researchers frequently use collections of functions and data, called software packages (also referred to as libraries), most of which depend on each other, to simplify the code used for analysis. In other words, the code within these packages uses functions or data from other packages to work. These dependencies can be version-specific, meaning that the analysis relies on a specific version of a crucial package to behave as it did when the original results were computed. In some cases, they require a specific version to run at all, since sometimes the new package version will be incompatible with the older script. The infrequency of software updates to certain packages is a substantial issue. A study in 2018 by Kula et al. found that the 4,600 well-maintained software projects they studied on GitHub used 2,700 package dependencies. In addition, 81.5% of the software projects relied on outdated dependencies [23]. Through correspondence with developers, the authors found several reasons why the developers did not prioritize updating package dependencies [23, p. 408-409]. Developers stated that even when informed of a dependency on a package with a security vulnerability, they either did not believe it had any significant effect on the major uses for their program, or they felt that migrating to a new, less vulnerable dependency took too much effort. Some expressed difficulty explaining vulnerable software dependencies when speaking to customers. They reported that stressing the importance of addressing these dependencies to their customers was also difficult. However, most developers the authors contacted were simply unaware that their program relied on a vulnerable dependency.

The lack of awareness and motivation to address out-of-date software dependencies reduces the reproducibility of research reliant on them. In 2014, computer science researchers at the University of Arizona sought to measure the extent to which software from a research project was buildable and usable [11]. One of the major issues they encountered involved finding software versions consistent with the

original research, and they even stated that the original authors of the research were often unsure which versions they used when producing a result [11, p. 16]. In addition, some of the original software used for research required an obsolete or unavailable dependency [11, p. 18].

Fortunately, capturing metadata or provenance can aid reproducibility. By understanding the conditions that affected a script's behavior, researchers can recreate identical conditions for successful reproducibility of the result if the original versions they used are available and accessible. "Data provenance" refers to information describing the origin of a piece of data and the transformation processes to reach its current form. In the context of workflows, which refer to repeated patterns of activity using software, "data provenance" refers to the collection of information about the origin of the result, namely details about the process and data used to produce the result [12]. As demonstrated above, reproducibility is difficult or impossible when the data processing details, such as the computing environment during the analysis, are unknown. Without understanding, locating, and utilizing the corresponding software and hardware components, such as the original scripts, data, operating systems, language version, and more, attempts to reproduce results are hindered. This is due to the interconnected ways in which these components affect computational practices and functionality of code used for analysis. Therefore, in the context of this thesis, "data provenance" is essentially the record of a script's execution, including enough information to recreate the conditions of that execution and verify results.

This thesis will focus on reproducibility in the context of R, a popular programming language used for many statistical computing and data visualization applications.[1] R developers refer to R as an 'environment' because it is a system of statistical software tools with opportunities for users to extend usability through third-party development of packages written in the R language. While this sig-

[1] `https://www.r-project.org/about.html`

nificantly extends R's capabilities, it also creates issues for reproducibility due to the sheer number of third-party developed packages used for data analysis in R, many of which are interdependent. The two largest sources of packages for R are the Comprehensive R Archive Network (CRAN) and Bioconductor. As of February 2023, there are 19,164 active packages on CRAN.[2] CRAN has a strict policy that R package developers must follow for their packages to remain active and as a result, packages that are not maintained can be archived and become unavailable for public use.[3] Proper maintenance involves upgrading packages when a new version of R is released or if the package "shows any warnings or significant notes," which can occur if dependencies update. For context, R typically releases a new version every spring, with patch releases following as needed.[4] If a developer's package upgrade could significantly disrupt other package functionalities, CRAN must approve that update well before its release to the public. The strict policies ensure that packages on CRAN are well-maintained and safe to use for statistical computing in various ways; however, the functionality of data analysis scripts is often dependent on the continued maintenance of many packages managed by third-party developers. CRAN maintains an archive of 20,840 older and unmaintained packages, including older versions of packages still active. Therefore, collecting data provenance for data analysis in R is crucial when trying to re-execute older scripts that may rely on dependencies no longer supported in the current version of R.

1.2 Goals

To address this reproducibility crisis in the context of the R programming language, my first two goals for this project were: (1) to discover or develop a usable method for reproducing results from R scripts where reproduction of results in the current computing environment was not possible and (2) use this method to make some

[2] `https://cran.r-project.org/web/packages/`
[3] `https://cran.r-project.org/web/packages/policies.html`
[4] `https://developer.r-project.org/`

preliminary conclusions about how changes to the computing environment affect script executions and results differently. My approach for the first goal involved recreating the original computing environment by using the data provenance of a previous script execution to identify which R version and relevant packages were employed to produce the results. Emphasizing usability, I wanted my method to be accessible for researchers without programming experience and thus, part of my tool would parse a data provenance file to recreate the computing environment without requiring user expertise.

Once I developed this method for reconstructing specific computing environments, I wanted to investigate the significance of differences between the original execution and execution in the current computing environment. My initial step involved a series of preliminary experiments conducted manually to determine if any patterns seemed to exist. Then, I created a tool to automate this process, which I will elaborate on in later sections of this thesis. My vision for the basic architectural framework of this component was a script that would collect data provenance on two executions of a script in the current and reconstructed computing environments and then compare them, trying to identify both major differences, such as the code not executing, returning an error, or significantly different output, and minor differences, such as a differing function call, or variable value changed slightly. Thus, this comparison was conducted at two levels, observing both changes to output data and how the scripts execute.

1.3 Initial Limitations and Constraints

To focus on a specific aspect of the reproducibility issue, I established a few strategic constraints on the kinds of scripts and computing environment changes I would study. I centered my project around reproducibility issues while using the R language for data analysis. While reproducibility is an issue when using any data

analysis method or software, I chose to focus on R because it is one of the most popular statistical computing languages, especially in academia. The language's use spans from its release in 1994 to the present day [8]. There are myriad publicly-accessible archives of R scripts that researchers have used for analysis. Furthermore, CRAN houses older versions of the R programming language and numerous packages, meaning the components to recreate computing environments are readily available.[5]

Since I relied on CRAN for available packages and their older versions, all the scripts for my testing and development could only depend upon those packages. However, I never went out of my way to enforce this constraint since all the package dependencies I encountered, both the current release and previous versions, were available on CRAN. I restricted the forms of output data for analysis, prioritizing scripts that produce numerical data to simplify the comparison of results. Also, the software I used for data provenance collection and comparison did not support raster data, so no sample scripts processed this type of data as input. In addition, the rdtLite package I used to collect provenance required a minimum version of R from 2018, so my research did not explore R versions before that date. Finally, I conducted all my research on one machine, so I did not explore how different computer architectures and operating systems affect reproducibility. Further discussion of these limitations will appear in Section 5.2, providing possible avenues for project expansion to address them.

1.4 Contributions

This project explores avenues for using data provenance for computing environment reconstruction, contributes two tools intended to demonstrate the potential of data provenance in addressing reproducibility issues, and makes some preliminary conclusions about how changes in various aspects of provenance may affect the execution

[5] `https://cran.r-project.org/src/contrib/Archive/`

behavior of a script. After researching methods for building a simulated computing environment from the environmental variables stored in a provenance file, I developed a script that parses the provenance file and creates a folder containing all other files necessary to build and run a Docker image simulating the original computing environment. Experiments with this tool revealed how different changes to the computing environment affect script behavior. A second development phase adapted a tool that compares provenance files from two executions to suggest which changes to the provenance have the highest potential to significantly affect script behavior and identify changes the user should make first in order to reproduce results.

1.5 Thesis Structure

Chapter 2 will dive deeper into the fundamentals of R data analysis and how the reproducibility issues directly relate to the language's various attributes. From there, I will overview related work, including tools to collect and use data provenance in R. In addition, I will explore approaches to recreating computing environments and computational research practices to make results more reproducible. In Chapter 3, I will discuss my approach to the issues introduced above, including more details about my progress in meeting my goals. This chapter will also illustrate two designs for the computing environment reconstruction tool. I will also describe my experimentation to understand how different computing environment changes affect script behavior. I will report and discuss my results at the end of Chapter 3. In Chapter 4, I will introduce my second development phase, adapting an existing data provenance comparison tool. I will discuss the significance of these contributions and possible future work in Chapter 5.

Chapter 2

BACKGROUND

This chapter provides a more in-depth explanation of data analysis in R, includes detailed definitions of commonly used terms, and illustrates some typical use cases for the language. Then, it explains how reproducibility is uniquely difficult for researchers who use R. Finally, the chapter overviews a selection of significant projects addressing the issues noted in Chapter 1.

2.1 Data Analysis In R

The R programming language's designers, Ross Ihaka and Robert Gentleman, sought to combine the strengths of two pre-existing programming languages at the time of its creation: S and Scheme [19]. Specifically, they aimed to merge Scheme's evaluation strategy, which in this context means the rules for evaluating expressions, and S's syntax.

Like Scheme, R is an interpreted language.[1] The interpreter, or program which executes code, reads each expression line-by-line and executes the instructions on that line before moving on to the following line [2]. There are two main methods for passing instructions to the interpreter: in a script or through the R console.

[1] Although some Scheme implementations support a compiler, most programming is done using interactive REPL command-line interface, like R

Users can enter expressions through the R console and see how the R system interprets them. Typically, users type expressions into the console to learn about the R environment, such as the value of a variable. The console is helpful for debugging; however, directly typing expressions into the console line-by-line becomes tedious when the applications for R become more complex. The other method for executing R code is writing an R script: a collection of R expressions written in one file. To execute an R script from the command line, most users employ the Rscript command, which interprets and executes each line of code from the script while maintaining access to the system's standard input, the R console. The second method is sometimes called "batch mode."

R supports numerous built-in functions for data interaction, analysis, and visualization [15]. These built-in functions make the language easy to use without much programming background, increasing the usability for non-technical researchers who want to do data analysis.

Most R programmers use RStudio, an open-source integrated developer environment (IDE),[2] which provides a more approachable interface and is the preferred IDE for many R users. RStudio's interface introduces many nifty tools, including multiple tabs to display different files, an R console for directly interacting with a session, a graphical display of variables in the environment, and a searchable help section for seeking information about functions, packages, and related files. RStudio assists with the installation of R packages, which expands the functionality of R for researchers with less programming experience and provides a graphical user interface for installation that is more accessible for those unfamiliar with the command line.

A **package** is a collection of related functions, help files, and data files for simplifying different tasks and functionalities. For example, ggplot2 is a frequently used

[2]As of July 27th, 2022, developers changed the name of RStudio to Posit (see https://posit.co/blog/rstudio-is-becoming-posit/)

package for data visualization.[3] Instead of each user writing all their functions from scratch, R provides this handy way for exchanging already-made functions and data between users. Users can call the "ggplot" function to create many different types of plots of their data. Some packages come with an R installation and others are available through public repositories, like the Comprehensive R Archive Network (CRAN). RStudio simplifies CRAN package access through the "packages" panel that streamlines remote package installation. Anyone can make a package in R; however, there are strict criteria for which packages are available in public repositories. To use an R package, the user must install it into a local library and then load it into their current R session. Two different R packages could contain functions with the same name, so by only loading the packages needed for a project, the risk of this conflict creating issues is minimized. The inclusion of packages establishes a middle ground in complexity for researchers, enabling them to use building blocks created by other developers in their research methods while maintaining an understanding of the underlying processes the computer is performing [28].

Unlike other tools and research methods like commercial software, which do not provide detailed information about what is going on behind the scenes, R is open source. This advantage means the source code is freely available for perusal and use, allowing for the expanded development of these packages. For example, the Rccp package, created by Dirk Eddelbuettel and Romain François, streamlines the integration of C++ code in R [14]. Rccp allows R programmers to use more computationally-complex functionalities written in C++ to boost performance and extend functionality from external C++ libraries for use in R. This type of third-party development integration would not be possible if R was not open source because developing useful extensions without seeing and understanding the source code produces major limitations on the functionality and accuracy of the extension.

To summarize, R is a popular statistical programming language. Using third-

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

party developed packages extends the capabilities of the language and allows researchers of different technical backgrounds to use more complex data and statistical analysis methods. However, as discussed in Chapter 1, the extensive use of these independently developed packages creates issues for reproducibility.

2.2 Data Provenance in R

While the issue of reproducibility is not unique to a single discipline or programming language, the flexibility and usability of the R language hinder reproducibility efforts in two main ways. Firstly, because packages developed by third parties are incongruently updated, some packages may become incompatible with the new versions after updating. CRAN manages this by either archiving the dependent packages or withholding approval for the updates.[4] Typically, CRAN corresponds with developers via email warning about impending changes to their package dependencies and the developer will be instructed to make changes. This is also how CRAN deals with R updates that break package code. If a developer fails to comply with these instructions, their package is archived. As time passes, the conditions that produced a result, including specific package versions, become inaccessible without reverting to previous software versions. Therefore, recording details about the conditions of the computing environment when researchers run scripts—the data provenance—is imperative for reproducibility. Secondly, because R is an interpreted language with multiple ways to execute code, data can be altered by both the script and other expressions passed into the console. Thus, keeping a copy of the original script alone may not record all of the data changes that produced a result. In this next section, I will explore data provenance in R in more detail.

Researchers classify provenance in different ways. In 2008, Davidson et al. introduced prospective and retrospective provenance as terms for classifying data provenance in the context of workflows [12]. Prospective provenance refers to "the steps

[4] `https://developer.r-project.org/`

that need to be followed to generate a data product," while retrospective provenance describes the "steps that were executed as well as information about the execution environment" [12, p. 2]. In other words, prospective provenance defines the execution steps as they are planned by the programmer, while retrospective provenance details information about a specific execution: what actually happened.

Another method for categorization looks at how we can store provenance information. In "Replication of Data Analyses: Provenance in R", part of "Stepping in the Same River Twice: Replication in Biological Research", authors Emery Boose and Barbara Lerner introduce the concepts of static and dynamic metadata [7]. Static metadata describes the input data and computing environment of an execution. This type of metadata can be stored as items in a file because each piece of information represents a value-variable pair. For example, there is only one operating system, one version of a language and one version of each package for a specific result. Dynamic metadata describes how data flows through computational expressions [7, p. 198]. Both pieces of data provenance capture relevant information for reproducibility. When seeking to replicate a computing environment, static metadata becomes more relevant. Meanwhile, expressions may call functions from different packages and have different behavior over time as versions change. Determining which changes to the computing environment affect a result requires understanding how data flows through a series of expressions. Here, dynamic metadata becomes imperative.

Within the umbrella of data provenance, we can also identify language-level provenance. This class of provenance describes how data is manipulated or changed by a programming language, such as R [9]. This is opposed to other domains of data provenance that record how data queries or file modifications may affect a piece of data. This thesis uses the collection of data provenance from the execution of a script and concerns language-level provenance. This is because language-level provenance relates most significantly to how different packages used in the computing

environment produce results.

In addition, this thesis concerns the reconstruction of computing environments and then the identification of how computing environment changes affect results. Therefore, Boose and Lerner's classification system is more applicable in this context, especially since static metadata describes the information needed to reconstruct a computing environment while dynamic metadata reveals how those changes affect script behavior.

2.3 Related Work

This section introduces various software designed with R reproducibility in mind. Section 2.3.1 introduces the most well-established tool for collecting and using data provenance in R: E2ETools.

Then I will focus on three main approaches to address reproducibility. The first approach involves recreating computing environments that mimic the original environment at the time of execution to reproduce the results of data analysis. Virtual machines and Docker containers are two popular methods for this approach. A second way to address issues with reproducibility is tools to organize and facilitate the distribution of scripts, including workflow applications and the DevOps methodology. The third approach will introduce two version control tools for R. The section will conclude with a brief description of Reproducibility as a Service, which is a tool developed for creating computing environments when data provenance about the original execution is inaccessible.

2.3.1 E2ETools

E2ETools is a collection of packages to collect, process, and visualize data provenance as well as provide an interface for other developer use of prov.JSON files [24]. I use the E2ETools suite of packages to collect and extract data provenance for my

reconstruction of and experimentation with different computing environments.

rdtLite

The R language has no built-in functionality for tracking data provenance [27]. However, researchers at Harvard University and Mount Holyoke College created a package called rdtLite which facilitates the collection of data provenance from both script executions and interactive console sessions in R [24]. The package captures information about the script, input files, output files, and plots, as well as the computing environment upon execution (Figure 2.1). In addition, rdtLite documents an execution trace of top-level R statements, collecting information about the executed expressions, variables created or used by those expressions, and additional details about their values [24, p. 4]. These values are written to a JSON file which follows an extension of the PROV-JSON standard [18]. This extension defines the addition of fine-grained provenance in the file.[5] The prov.JSON file includes additional information concerning fine-grained provenance, which Lerner et al. define as "the data used on that line and any data computed by, or object created by, that line" [24, p. 2]. rdtLite also makes copies of the scripts and input/output data from that execution to preserve the specific version to produce those results. The names of these copies are included in the prov.JSON file.

provParseR

provParseR allows the user to parse a prov.JSON file and provides an API for extracting data provenance. Calling the prov.parse() function returns a "ProvInfo" object which a developer uses to obtain information about the input files, output files, environmental variables, name and version of libraries used, and the name of each function used from each library. My tool for computing environment reconstruction uses provParseR to extract information about the R version and packages

[5] see https://github.com/End-to-end-provenance/ExtendedProvJson/blob/master/README.md

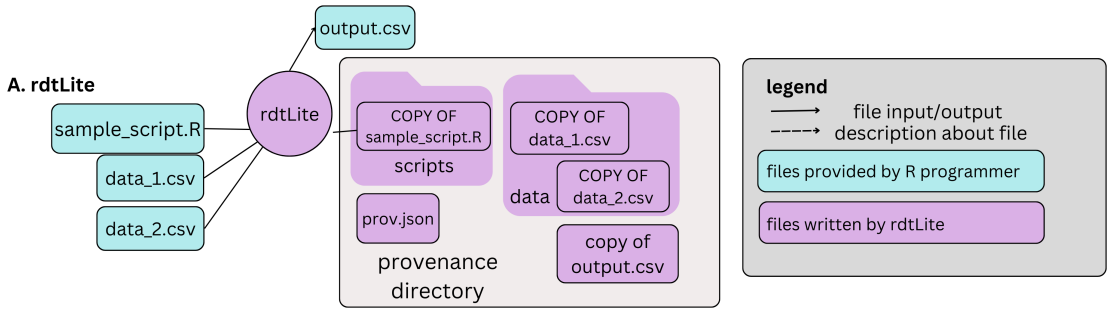


Figure 2.1: Execution flow using rdtLite

from a data provenance file.

provExplainR

provExplainR compares two prov.JSON files, presenting a readable report of the differences between the two executions. This package can identify all changes in the computing environment and data inputs to determine what might cause discrepant results. The original version of provExplainR identifies these differences in the scripts, input files, computing environments, and provenance tools from each execution by extracting data provenance with provParseR and comparing the ProvInfo objects. The method divides the provenance into subsections based on each aspect of the provenance listed above and prints a statement for each aspect detailing whether they are identical or different. provExplainR also allows the user to save the output into a text file, though by default the report is displayed via the R console. Part of my thesis extends provExplainR to also compare the script behavior for each execution to suggest which other provenance changes affect behavioral differences.

provViz

provViz is a visualization tool that displays a color-coded provenance graph based on a given prov.JSON file. These provenance graphs consist of two types of nodes: data nodes and procedural nodes. The edges between these nodes indicate the flow

of data between expressions (Figure 2.2). An edge from a data node to a procedural node shows that the expression in the procedural node used the data in the data node. Similarly, an edge from a procedural node to a data node signifies that the procedure created that piece of data. There are four types of data nodes. "Data" type data nodes (purple) represent data stored in variables. "Exception" type data nodes (red) represent warning and error messages. "StandardOutput" type data nodes (dark orange) represent messages to standard output, usually the R console. "File" type data nodes (light orange) represent files imported into the script. The provViz package has a variety of applications, including being a visual aid for presentations and facilitating a deeper understanding of what a script is doing for debugging purposes. With provenance graphs, a user may view input/output files, plots, and source code of an execution and search for nodes using both name and type. In the context of reproducibility, provViz allows visual comparison between two similar executions to help identify where behavioral divergences occur and which procedural nodes might be causing those divergences. Developers can also use provViz to understand the relative execution time for different expressions by sorting by them. All nodes and edges are stored in data frames returned by provParseR, and scripts which compare these data frames glean the same information that a user may from observing the graphs visually.

2.3.2 Recreating Computing Environments

As mentioned previously, software version updates can alter an older script's behavior, causing different outputs and/or errors. Sometimes the only way to execute an old script is to recreate the original computing environment it was developed in. This section will detail approaches to computing environment reconstruction.

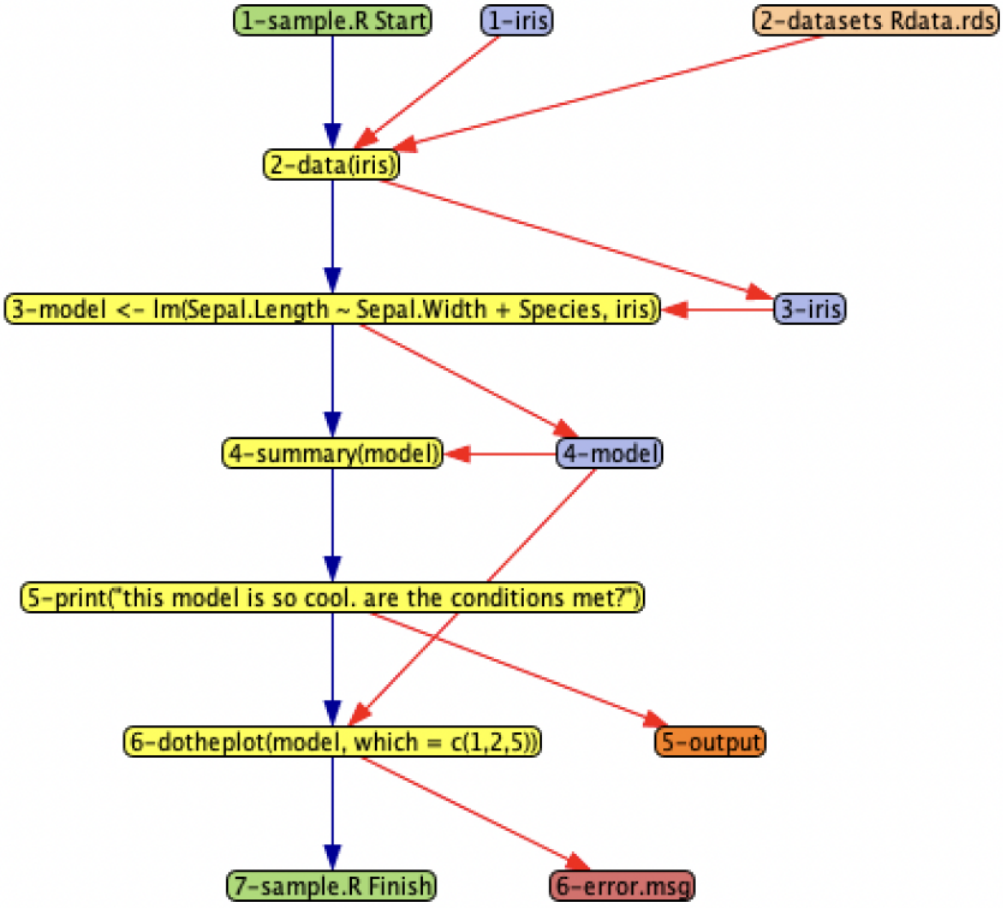


Figure 2.2: Provenance graph created by provViz

Virtual Machines

Virtual machines provide a method for reconstructing computing environments. One approach, whole system snapshot exchange (WSSE), was developed by Dudley and Butte in 2010. This involves capturing all the information about a computer system and storing it in a sharable format [13]. Their approach is meritable because it avoids all discrepancies caused by software component version changes in the languages, packages, and operating systems. The authors propose the utilization of cloud computing technologies, which involve detaching the computation and data from one piece of hardware and sharing them among disparate clusters of hardware systems, to avoid the issues with transferring large amounts (gigabytes or terabytes!) of data [13, p. 1182]. In 2018, Bill Howe at the University of Washington contributed further to researching the approach of using cloud computing to support virtual machines for reproducibility [16]. Howe advocated for publicly-funded cloud computing infrastructure for researchers to use, suggesting that they submit snapshots of their virtual machine used for conducting research, making it publicly-available and cited in related papers [16, p. 2]. However, using virtual machines for this work presents issues with scalability when the technologies are used at a much larger scale. Furthermore, this approach does not require researchers to document dependency use, making the information less extractable for other uses besides constructing the virtual machine. Since each study would have one or more virtual machines, using tools from multiple studies in one project may not be feasible or even possible [5, p. 73].

Docker

Docker is an open-source software tool that allows developers to share and run an application in a somewhat separate environment called a container.[6] Each container contains all components needed for successful application execution. Multiple con-

[6] `https://docs.docker.com/get-started/overview/`

tainers can run separately on the local machine without interfering with it. To build a container, Docker uses a read-only template with instructions called an image. Users can customize images using a Dockerfile, adding components to tailor the container to their needs. The Docker architecture follows a client-server design, consisting of two main components: the Docker client and the Docker daemon. The client facilitates communication between the user and the Docker daemon, passing messages that use the docker command; the client can also communicate with multiple daemons. The Docker daemon processes commands from the client and manages all the images, containers, networks, and volumes. Designers establish Docker's isolation with separate sets of namespaces—fields that computers use to identify and reference various objects—for each container.

Docker provides a more lightweight version of a virtual machine because while the container is isolated from the local machine and contains all components necessary for application functionality, it still uses the Linux kernel from the host machine.[7] This means that on Linux machines, Docker performs significantly better than most virtual machines on a typical desktop computer [5, p. 74].[8] Using Docker containers, researchers can capture all the dependencies and conditions needed to produce a result and distribute the components for reproducibility with more ease. In addition, users can continue using familiar data analysis tools for research while still working towards a reproducible result. However, Docker is not without limitations. It requires 64-bit host machines and cannot run on older hardware. Furthermore, it does not facilitate complete virtualization, meaning it cannot simulate multiple hardware types. This prevents the resolution of reproducibility issues at the hardware level. Critically, the larger scientific community has not widely adopted Docker; almost all its users are in computer science-related disciplines [5, p. 77].

[7] For non-Linux machines, the container uses a small Linux virtual machine [5, p. 75]

[8] A typical desktop computer can run only a few virtual machines at once but hundreds of Docker containers

Rocker: Docker for R

The Rocker project provides Docker containers specifically for R programmers, with the primary goal of rapid deployment of R environments to remote servers and private/institutional hardware, ultimately scaling analysis procedures by eliminating the time it takes to ensure code runs smoothly in a variety of different environments [6, p. 527]. Notably, the Rocker project also integrates Docker usage with the RStudio Server integrated development environment (IDE), allowing users to access Rocker with a more usable graphic interface instead of the terminal.

Ultimately, Rocker supports R users who want to use Docker for distribution and reproducibility for containerized projects. Rocker expands the number of R users who can benefit from containerization in software deployment, public sharing of computing environments, prepared environments for teaching, and reproducibility [26]. However, this approach does not directly require data provenance collection and use from script executions, opting instead to share containerized applications and environments. The use of Docker in academic and non-technical fields is expanding, though there are few tools that assist in Dockerfile creation for creating snapshots of research. Furthermore, these solutions only address the reproduction of research conducted now and cannot facilitate the reproduction of results whose computing environment was never containerized.

2.3.3 Script Organization and Distribution

Another category of approaches for resolving issues with reproducibility involves software products and design methodologies which keep track of what components are needed for a specific execution [5].

Workflow Software

Workflow software is designed for the integration of multiple software tools. Despite being well-designed and relatively easy to use, these systems are not widely

adopted because the efforts to create reproducible workflows are not rewarded in contemporary science. The tools are largely commercialized and privatized, and thus are not open source. Furthermore, these tools are designed to be generalizable and cannot meet the needs of every individual user the way open-source data analysis tools can [13]. Kepler is an example of a workflow system, released in 2004 and designed to simplify various data exploration and analysis procedures for scientists while capturing information about their workflows for easy reproducibility [3].[9] Another example was Taverna, released in 2006 and designed for bioinformaticians to integrate various software tools and databases for their analysis [17];the Apache Software Project, which maintained Taverna, retired it in February 2020.[10] More recently, new workflow management software for scientific research, such as ProjectFlow, focuses on specific applications and fields. There are no widely-used generalized workflow management tools today.

DevOps

A distinct approach to the issue of reproducibility in the scientific computing field has emerged more recently, sometimes called DevOps or Development and System Operation, which involves the documentation of install paths for relevant dependencies and providing scripts that automate system set-up [10]. One common example of the DevOps-style practice are Makefiles [10, p. 8]. The scripts in this approach address all attributes of the computing environment from the operating system up, including language versions and dependencies. Proponents of this approach emphasize that it requires no new tools; simple package management and shell scripting tools are sufficient. Several tools make this approach simpler. The aforementioned Docker, one of these tools, stands out because of its growing popularity and online user support.

[9] As of 2023, Kepler is a SaaS AI platform which automates workflows, allowing users to utilize AI technology. (see https://www.softwareadvice.ie/software/110520/kepler)

[10] `https://incubator.apache.org/projects/taverna.html`

2.3.4 Version Control

Instead of attempting to capture the entire field of components necessary for execution, some solutions provide easy ways to switch between specific software versions. While these solutions do not scale well when projects are managing hundreds of packages and many other computing environment components, they do allow targeted control of specific software versions for small projects.

renv

renv, previously known as packrat, is an R package available on CRAN which supports reproducibility by providing each project with a private package library.[11] Each project-specific library is isolated and installable, allowing a user to install different versions of packages for various projects and transfer all the necessary dependencies across different machines. While renv provides a simpler approach to package dependency management, it does not address other components of the computing environment, like R language versions, and requires its use during the project's development to facilitate the reproducibility of said project.

RSwitch

RSwitch[12] is a menubar application specifically for macOS whereby users can easily toggle between different versions of R on their local machine.[13] In addition, each version of R installed creates an isolated package library for that R version. However, because each version of R is installed on the local machine, certain components of external software required for R to run may be shared by each R version. This lack of isolation creates issues when installing versions of R that were released at a similar time.[14] Notably, RSwitch was archived by the developer on June 2nd, 2022 because

[11] `https://rstudio.github.io/renv/articles/renv.html`

[12] Not to be confused with "RSwitch the National E-payment switch of Rwanda." `https://rswitch.co.rw/`

[13] `https://github.com/hrbrmstr/RSwitch`

[14] This issue with RSwitch will be further explored in Chapter 3, "From rdtLite to Docker."

another developer created an improved and more broadly usable implementation called “rig”.[15]

rig

rig is another R version manager developed by Gábor Csárdi and initially released in 2021. This application fulfills a similar role to RSwitch, however, it is usable for Windows and Linux as well as macOS. It also creates a menubar app for macOS, allowing for ease of use.

2.3.5 Reproducibility as a Service

Reproducibility as a Service (RaaS) is a system designed by Joseph Wonsil at the University of British Columbia to address the issue of retroactive reproducibility, which he defines as “reproducibility for a computational experiment that no longer has access to a sufficient computational environment” [29, p. 3]. Many tools described above require extensive knowledge of the original computing environment. Unfortunately, there is a plethora of scientific knowledge and results for which that information is unavailable. Focusing on this issue, RaaS “attempts to fix common retroactive reproducibility bugs” and “uses static analysis to collect the information needed to re-create a computational environment within a Docker image” [29, p. 3-4]. After constructing the Docker image, it executes the original analysis and uses rdtLite to gather language-level data provenance for verification. Then, a user can distribute the Docker image through Docker or as a downloaded, compressed file. According to Wonsil’s evaluation, RaaS could resolve 85% of library errors and 86% of working directory errors. Furthermore, RaaS was able to increase the success rate of reproducing scripts from 11.3% to 31.0%. Of the 69% that still encountered errors, 75.3% of scripts had different errors than when they were run without RaaS, demonstrating that RaaS was still successful at some error resolution [29, p. 4]. Ul-

[15] `https://github.com/r-lib/rig/`

timately, RaaS successfully attempts to address the instances when data provenance is not available.

Chapter 3

APPROACH

3.1 Introduction

This thesis' approach to observing how changes to the computing environment affect the success of reproducing results had three main steps. First, I recreated the computing environment from when a script was originally executed. Then, I mutated the computing environment and observed how environmental changes impact the script's results. Finally, I extended E2ETool's provExplainR to pinpoint the cause of discrepancies in their script's output and prioritize the most significant ones for a user to address. This chapter details the work to achieve these steps.

3.2 Environmental Reconstruction

3.2.1 Early Attempts

In preparation for writing my thesis proposal and presentation, I researched and collected several potential tools to facilitate the reproduction of computing environments based on data provenance. I wanted to prioritize simplicity, employing tools with the most straightforward approach and capabilities in the hopes that later work could build upon my more narrowly-focused project. In addition, I hoped

the methods I derived could be used in the meantime by those with non-technical backgrounds. Thus, I explored the capabilities of the R package "packrat" in terms of computing environment reproducibility.[1]

packrat

The R package "packrat"[2] is a dependency management package that aids users by keeping track of different package versions required for a given user's various R projects.[3] As explained above, the pace at which R and its packages update often creates issues for reproducibility when packages become incompatible with each other or the newest version of the R language. Thus, users may require an older package version to run a script. Often, users are working on multiple projects with many scripts that may require different versions of the same package. To address this, packrat allows a user to create independent package directories to house all the necessary package versions for a given project.

When installing a package in R, the most popular function is "install.packages()". A user may pass this command into the R console with the package's name as a string for easy installation. Alternatively, using RStudio's interface, clicking "packages" → "install" and then typing the package name automatically passes the same "install.packages()" expression into the R console. However, unless a user is installing packages from their local machine that they downloaded as a package archive file, there is no way to specify the version of that package. Often, users do not want to manually download a package archive file and then install it into their package directory, so they use "remotes::install_version()", which allows for the specification of the package version using the 'version' parameter. Another method familiar method for this, "devtools::install_version()", is the same function exported from

[1] For a refresher on "packrat", read Section 2.3.4.

[2] As stated in 2.3.4 renv is a newer package developed by the same authors which is the focus of all new development. However, packrat is still being maintained. At the time of my experimentation in the Fall of 2022, I used packrat exclusively for this section of my work.

[3] `https://rstudio.github.io/renv/articles/renv.html`

the "remotes" package with a small extension.[4]

While experimenting with packrat, I successfully created a few private package directories. I intended to use "remotes::install_version()" to install specific package versions to the private package directories created by "packrat". However, I had some issues using "remotes" along with packrat's interface. By installing a specific package version as an archive file, I could install any package version using the "packrat::install_local()" local function.

Packrat successfully created isolated package directories for facilitating version control for separate R projects; however, it did not address version control at the language level. In other words, "packrat" could not, as far as I could tell, be used to install specific versions of R for different projects. Since it was crucial to include the R version in the computing environment reconstruction, I decided "packrat" alone was not the method for this project. Nevertheless, I did explore its use in tandem with another tool that managed version control for the R language, RSwitch.

RSwitch

RSwitch is a menubar application that facilitates the easy installation of and toggling between different versions of R. Each version of R has a private package directory and a user can manually install specific versions of packages into each directory for each language version. I identified three periods where I wanted to create an environment using specific R versions and then compiled a list of packages I needed, along with their dependencies. Using CRAN's archive of previous package versions, I cross-referenced the range of dates when these R versions were current to find the corresponding package versions during that period. This process was incredibly tedious because I was installing the packages manually to specify exact package versions and my test script, which used "rdtLite", had many package dependencies. I kept track of all the dependencies and their versions with a spreadsheet. For

[4]They differ from the "remotes" package functions because they use an ellipsis to make sure all dotted arguments are used. (see RStudio help page, ?devtools)

each installation, R would return the missing dependencies, which I would log into the spreadsheet. From there, I would research which version of that dependency corresponded with the relevant R language version as the current version. I would attempt to install those dependencies and the cycle would continue. Eventually, I could install all the package version dependencies for rdtLite and created a complete computing environment for a computer active during February-April 2020 running R 3.6.3. From there, I attempted to repeat the process, using R 3.2.1, the current version of R from June-July 2015. However, I quickly encountered an unavoidable issue. rdtLite, the tool I planned to use for data provenance collection to verify my tool's work and eventually use for provenance comparison to understand the significance of changes, was incompatible with R 3.2.1. The earliest version of rdtLite was released in October of 2018, so it would not run on any version of R from before that. This created another limitation for which scripts I could experiment with, since I could not use rdtLite if my script execution was from before that period.

In response, I sought to repeat the process with R 3.5.1, the version from the rdtLite release date. Using RSwitch, I installed R 3.5.1 and started installing the relevant package versions, repeating the previous method. Yet, I encountered another dependency issue. The installation of R 3.5.1 disrupted the functionality of R 3.6.3. I was limited to what I could see in the menubar interface, which displayed that R 3.6.3 was now incomplete (Figure 3.1). I speculated this was because the installation of an R version close in age to the already installed version may overwrite some files that both required. Because of this issue and my doubt that I could implement a more user-friendly program to facilitate this method of package installation, I opted for another method. Further research after choosing to change directions led me to discover that the RSwitch tool was archived.

During the same investigation, I discovered 'rig': another tool that allows the user to switch between different versions of R. I chose not to experiment with this tool for two reasons. The first is that by the time I discovered it, I had already

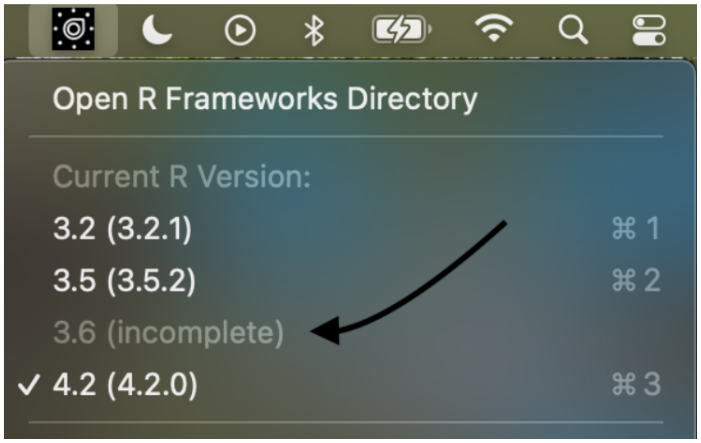


Figure 3.1: Menubar drop down showing R version incompleteness

devoted a lot of time to the method I will discuss next. The second is that the tool is relatively new and still has a lot of known issues and limitations which the author is working to fix. Overall, I felt that a disjointed approach to the reconstruction of a computing environment, namely attempting to install the correct R version and the correct package versions using different tools, was not as simple as it appeared and may not lend much effectiveness to my work. From there, I opted to explore Docker as a solution for reproducing computing environments.

3.2.2 Using Docker

In my project proposal feedback, faculty members mentioned that Docker was the primary solution computer scientists employ to facilitate reproducibility and mitigate dependency issues. Feedback also concluded that Docker may be too technically advanced for use by researchers without much computer science experience. After the issues with packrat and RSwitch, I changed course, hoping to develop a tool that made Docker more accessible to non-technical researchers and used prov.JSON files for reconstructing computing environments. This section will detail the development of this tool.

To begin, I launched a more in-depth investigation into what Docker was, how it worked, and where a tool could help decrease its use complexity. Docker uses

"containers": lightweight, portable, and isolated environments that simulate computation from the operating system up. In other words, the user can design a Docker container to mimic an environment with any parameters they want, starting with the operating system as the base level of customization. Docker builds containers from "images", which provide a foundation for the container, including details about which operating system to use and what to do once the container is running. Docker developers provide many "base images" which users may alter using Dockerfiles. Dockerfiles build on top of a base image, create a custom image, and can instruct Docker to install external software for use within the container.

Equipped with this additional information, I developed a prototype script that parsed the prov.JSON file and wrote a Dockerfile to construct a customized image that Docker could build into a container resembling the computing environment described in the prov.JSON file. My script, which I called "createDocker.R", contained a function, called "createDocker", that took two strings: "prov.dir" for the provenance directory location and "docker.name" for the docker folder that would contain the Dockerfile. I chose to create separate folders because every Dockerfile needed the same filename to be recognized by Docker. Wrapping them in a user-named folder helped differentiate Dockerfiles. After verifying the provenance directory existed, the script parsed the prov.JSON file using provParseR and passed the data frame returned by provParseR into another function "generate.docker.file()". This function used the data frame to write a Dockerfile. In addition, "generate.docker.file()" created copies of any scripts or data in the original provenance folder and stored them in two folders: "<docker.name>/Analysis/" for the scripts and "<docker.name>/Data/" for data.

Recreating the computing environment in Docker involved selecting a base image that matched the R language version from the data provenance and corresponding R package versions. I chose to use the base image created by Rocker, using "rocker/r-ver:<r version from prov.JSON>". Also, the Dockerfile syntax provides

a command to execute R commands, which I used to install the remotes package and then the packages and their specific versions from the data frame using "remotes::install_version()."

I used the Terminal to build and run my images and the Docker Desktop application to keep track of each container. While I could successfully reconstruct the computing environment, I was not successfully executing any scripts within the environment. After Docker ran the image, it built my container and installed all the packages I required. Then, it exited with code 0, indicating that no errors occurred. However, I could not run scripts inside the container. I concluded I was not fully grasping how Docker containers functioned and chose to seek advice from individuals on campus who were more familiar with Docker.

In December 2022, I corresponded with Charles Romer, a lab instructor at Mount Holyoke College, at the recommendation of my thesis advisor, Barbara Lerner. I inquired about where my understanding of Docker was lacking and how I could improve my script to accomplish my goals. Romer recommended I try a slightly different approach regarding Docker. Instead of using the Dockerfile alone to create the entire computing environment, he suggested I use a simple Ubuntu Docker image with an "entrypoint" file. The entrypoint would execute after Docker builds my container and install all the necessary dependencies before running the script(s). My current method was feasible; however, it would require the user to re-run my script and generate a new Dockerfile every time they wanted to alter the computing environment components of the container. He also recommended I write and use a docker-compose.yml file to build my image, allowing me to create environmental variables and "volumes": dynamically shareable directories between the container and my local machine. Romer also provided an example system accomplishing the same tasks for the Python programming language, from which I modeled my implementation for R.[5]

[5] `https://github.com/cchaz003/docker_py_runner`

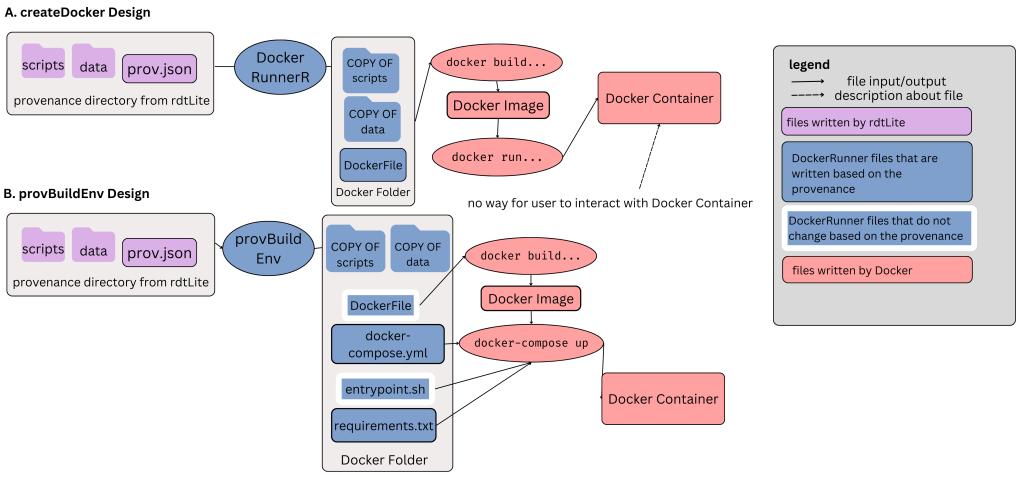


Figure 3.2: Diagram of execution flow createDocker.R (A), and provBuildEnv.R (B)

```
FROM rocker/r-ver:3.6.3
LABEL Maintainer="sfabrega"
WORKDIR /home
COPY entrypoint.sh /entrypoint.sh
RUN chmod 755 /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

Figure 3.3: Dockerfile example

Figure 3.2 summarizes the key differences between my original script using Docker and the updated implementation I adapted for the R language from Romer's "docker-py-runner". The updated system, seen in Figure 3.2.B, uses four files to create the Docker container. The first of these files is the Dockerfile, which uses an Ubuntu base image and instructs Docker to set the working directory as "/home", copy "entrypoint.sh" into the working directory, change the entrypoint's file permissions, and then run "entrypoint.sh" once Docker finishes building the container (Figure 3.3)

The second file, the "docker-compose.yml", provides instructions for running the image, including the setup of volumes and the time zone (Figure 3.4). This file also defines the command for executing the R script. The third file, "requirements.txt", is placed within the volumes directory to be accessible once the container is ready

```
version: '1'
services:
  rscript:
    image: saf-3
    volumes:
      -
/Users/seanfabrega/Desktop/workspace/scripts_for_ce_testing/pro
v_testing/prov_strings-as-factors/saf-3/volumes:/home
    environment:
      - TZ=America/New_York
    command: Rscript -e
"rdtLite::prov.run('Analysis/strings-as-factors.R', prov.dir =
'Data')"
```

Figure 3.4: docker-compose.yml example

(Figure 3.6). "requirements.txt" contains all of the required packages and their specific versions, corresponding with the information in the prov.JSON file. Finally, the "entrypoint.sh" file provides a series of instructions for Docker once the container is built, including setting "/home" as the working directory, installing the correct version of R, locating the "requirements.txt" file, installing all the packages listed using "remotes::install_version()", and executing the command declared in the "docker-compose.yml" to run the script (Figure 3.5). I called this tool provBuildEnv.

While the structure and some of the files in my implementation resemble Romer's original code, I implemented key adaptations for R. The R language uses external Ubuntu libraries to run, and Ubuntu does not maintain the availability of older versions of packages from the same source for installation .[6] Thus, if a newer software version becomes available, Ubuntu will switch out the older version for the newer version. When I attempted to install an older version of R, like 3.6.3, on the latest version of Ubuntu using "apt install", it reported that R 3.6.3 was unavailable. To fix this issue, I used a base image in my Dockerfile that corresponded with the R version from the prov.JSON file. Initially, I used the r base image from Docker; however, I ultimately decided to use the Rocker images because they are more

[6] https://itsfoss.com/apt-install-specific-version/

```
#!/bin/bash
cd /home
apt update
DEBIAN_FRONTEND=noninteractive apt-get --yes install build-essential
libcurl4-gnutls-dev libxml2-dev
FILE=./requirements.txt
PACKAGE_LS="c("
if test -f "$FILE"; then
 echo "$FILE exists."
 while read -r package version;
 do
  PACKAGE_LS+="'${package}',"
 done < "$FILE"
 PACKAGE_LS=${PACKAGE_LS::-1}
 PACKAGE_LS+=",'rdtLite')"
 echo ${PACKAGE_LS}
fi
 Rscript -e "install.packages(${PACKAGE_LS})"
exec "$@" # run whatever command is given for "command:" in
docker-compose.yml
```

Figure 3.5: entrypoint.sh sample

```
assertthat 0.2.1
bit 4.0.4
bit64 4.0.5
cli 3.6.0
colorspace 2.0-3
```

Figure 3.6: Snippet from requirements.txt sample

well-maintained. While this does limit the portability of my project, it avoids the constraint of inaccessibility to the R language through "apt-get".

Another R-specific issue is that there is no widely accepted way to use a "requirements.txt" file for installing dependencies. Instead of using pip3 (as Romer does in his implementation) I manually parsed the "requirements.txt" file using a while loop, extracting the package name and version information and passing that into the "remotes::install_version()" method for each line of the "requirements.txt" file. I did encounter an issue with installing R packages curl and XML, which are both dependencies of rdtLite, and ended up needing to install the Ubuntu libraries for them in the Docker container before installing the required R packages.[7]

When I began experimenting with provBuildEnv, I recognized a second functionality to implement for cases where the user does not have a complete provenance for an older script or wants to run a script on an older version of R. In this case, the user could set parameters in the function header so that instead of using a provenance file, provBuildEnv would use the version of R passed into the function. This loads the required libraries in the "requirements.txt" file using "install_packages()", which installs versions corresponding to the R version set in the Dockerfile. I used this new functionality for most of my experimentation since the ability to run the same script on different versions of R proved crucial for understanding how R version changes affect the provenance.

3.3 Observing Changes

3.3.1 Script and Data Collection

I anticipated the need for sample and test scripts as part of my experimentation and development phases. Thus, my first steps ensuing my proposal presentation were to

[7]Specifically, I installed libcurl4-gnutls-dev and libxml2-dev using apt-get and needed the –yes flag to permit the installation without interacting with the Docker container's virtual machine.

establish a criteria for which scripts I would be using and then seek out sample code from various online and local sources. Here is the set of constraints on scripts used for my project:

- Written in R
- Uses packages
- Creates outputs that are either numerical, graphical, and/or text
- Does not use excessively large data inputs, such as raster data
- Does not take more than 5 minutes to run
- Has some sort of date associated with its creation (upload date, creation date, etc.) either on the source website or in the script itself to know what version of R to use

The first two criteria exist because the tools I used to collect and process provenance are developed for R scripts which use third-party developed packages. To simplify the result comparison process, I focused on scripts that produced numerical (numbers or files with numbers), graphical (plots), or text (including .txt and .csv files) outputs. None of the scripts I used for experimentation or testing use raster data because it tends to be large and the E2ETools do not yet support provenance collection for scripts that process raster data.[8] Because the aim of my project was more of a proof-of-concept experimentation to facilitate reproducibility using provenance, I focused on simple scripts which did not take long to run and did not use excessive amounts of data. Finally, I wanted some reference points for computing environment reconstruction, which required some details, like specific dates, regarding the origins of the scripts and their associated output data to predict what the corresponding computing environment might look like. To keep scripts, data, and

[8]See Section 5.2 for more information.

provenance organized, I created a series of folders and spreadsheets which catalogued details about each component.

The script collection occurred in the fall of 2022 by Willow Kelleigh (‘25), a student at Mount Holyoke College. Kelleigh found fifteen scripts meeting the criteria from 2009-2022. Most searchable scripts from the Internet do not include data provenance-related information such as the date of their execution, or even corresponding results. Without this information, it is usually impossible to infer what the original computing environment that produced the results looked like; it was not helpful to collect those scripts. I instructed Kelleigh to source scripts from Kaggle, GitHub, the Harvard Forest archives, and any other sources which contained scripts meeting the criteria. Additionally, Kelleigh attempted to execute the script on their local machine and collected provenance for that execution, uploading a copy of that provenance and any output files from the execution along with the script itself. Five out of fifteen produced errors when executed in Kelleigh's local machine.

3.3.2 Measuring Significance of Change

Developing a Framework for Experimenting with R Version Changes

To begin the process of deducing the significance of various changes to the computing environment, I compiled a list of common modifications that could affect results (Table 3.1). I chose to focus on R version and package version differences since the specific mechanisms behind their disruptions to reproducibility were less understood. Scripts and input data changes often do modify the behavior of the script; however, those changes and their effects are typically easier to identify. Changes to the computing environment have the potential to cause a spectrum of behavioral discrepancies that are not always straightforward to understand or address. I sought to either create or find scripts that would either break or produce different outputs if one of these changes occurred, beginning with R version updates.

Fortunately, CRAN is transparent and upfront when an R language update could

Permutations Potentially Affecting Results
R version updates
Script changes
Input data changes
Package changes

Table 3.1: Permutations Potentially Affecting Results

cause disruption to a third-party developed package and publishes publicly-available reports for each update, containing all the relevant information about significant changes. I identified three major R language changes that required developers to revise their packages to prevent them from being archived. The first occurred when R 4.0 was released in December of 2020 when the default parameter, "stringsAsFactors", for data.frame and read.table switched from TRUE to FALSE [21]. Before this update, R automatically converted string-type data to "factors" when creating data frames and reading tables. For example, if a user imported a dataset about a student population with a column designating the state each student was from, R would automatically convert the data type of state names from strings to factors. Thus, the R interpreter uses a number to reference each state as opposed to the state's name. In other words, each string category is assigned a number, and R uses these numbers to access the data. Using factors over strings can be more efficient because accessing numbers is quicker than accessing strings. Storing numbers generally uses less memory as well. By transforming categorical variables from strings to factors, a user can easily access all the categories in that column. However, the numbers assigned to string-type categorical variables are locale-dependent and can differ within various R environments. Therefore, the automatic conversion was not easily reproducible. In other words, different executions of the same script might lead to each string category corresponding to different numbers. If a script used an integer constant to reference a certain level because that constant value originally matched the encoding of one category, a different encoding pattern would cause the incorrect category to be referenced and change the behavior of the script. Thus,

changing the default parameter settings created another reproducibility issue because a script might rely on this automatic conversion of categorical variables to factors.

In a report on the R 4.0 update, Kalibera et al. note, "a large number of packages relied on the previous behavior and so have needed updating" [21, p. 403], indicating that this change was and still may be a hindrance to reproducibility. In summary, this change significantly impacted results where the script used the data.frame or read.csv function —both ubiquitous operations in R—to parse a data table. Other packages, such as ggplot, also relied on these default parameters and the resulting changes affected the behavior of that function as well. Fortunately, after its release, many packages dependent on this syntax behavior were forced to update, as per CRAN's policies. However, individual scripts that rely on this syntax behavior are not strictly maintained and might produce different results.

The second significant change occurred when R 4.2.0 was released on May 31st, 2022. The newer version returns an error when if() or while() statements contain expressions of length greater than one, as opposed to previously where it would use the first element and report a warning [22]. Errors, unlike warnings, halt the execution of the script. Figure 3.7 is sample script which illustrates this distinction. After x and y are summed, x's length is greater than one and in R 4.2.0, the interpreter would throw an error and halt execution at line 4.[9]

This report also explains that comparisons using "||" or "&&" of expressions of length greater than one will only use the first element and return a warning. In R 4.3.0, it will return an error.

A significant library change I discovered but could not experiment with involved the base R package "stats" in 4.3.0.[10] R version 4.3.0 has plans for release on April 21st, 2023, and while R developers released the beta version for developer testing

[9]Only reporting a warning for this is very strange to programmers who use strongly-typed languages. I am unaware of any widely used language that would have been okay with this.

[10]`https://search.r-project.org/R/refmans/stats/html/family.html`

```
1  x <- 1 #type integer
2  y <- 1:10 # vector: y = [1,2,...,10]
3  x <- x + y # x = [2,3,...,11], vector
4  if (x == 2) { #length of x is > 1
5    print ("x is 2")
6  } else {
7    print ("x is not 2")
8  }
```

Figure 3.7: Sample code showing differing variable lengths and an if-statement

in March 2023, Docker has yet to release a R 4.3.0 base image. The "stats" package is an R "base package": a package that is part of the R language and automatically installed and updated with the language. The "family()" function allows users to specify details about a general linear model. If a script compares details about multiple models, the new version of R adds an extra component to those outputs, possibly changing the results of the comparison; this is a case where both the R version and the package version might alter the behavior of a script since R base package versions update with the R version. It can be hard to differentiate changes between these two aspects of the provenance. Table 3.2 summarizes the changes which occurred in these three R version updates.

Experimentation With Changing the R Version

The next step is to determine how the provenance of scripts that rely on the previous behavior of the R version would change when running on the newer version. To isolate the change in the default value of "stringsAsFactors", I wrote a script, shown in Figure 3.8, that declared a data frame using the default settings for the "stringsAsFactors" parameter, which varied in different R versions. Then I called levels() on a categorical variable column of the data frame, printing the values of each level in the column and the number of values in that category. I executed this script on my local machine using R 4.2.0. Using provBuildEnv, I created a Docker container using R 3.6.3 and collected provenance from the scripts' execu-

R 4.0	R 4.2.0	R 4.3.0
Default parameter "stringsAsFactors" in functions data.frame() and read.csv() changed from TRUE to FALSE		
	Using "||" (logical OR) or "&&" (logical AND) to compare expressions of length greater than one returns a warning	Using "||" (logical OR) or "&&" (logical AND) to compare expressions of length greater than one returns an error.
	If() and while() statements evaluating expressions of size greater than one return an error	
		"family()" in the "stats" package returns an additional output "dispersion" if fixed, or NA_real if free

Table 3.2: Summary of Relevant R updates

tions. Then, I compared the script output in both versions. Unsurprisingly, they were quite different. In R 4.2.0, when I did not set "stringsAsFactors" as TRUE, printing the levels resulted in a NULL output because the categorical variable column was not of type factor. However, when using R 3.6.3, the script printed the value R used to represent each category and the number of rows in the data frame in that category both when "stringsAsFactors" was and was not specified. When I used rdtLite's visualization tool, provViz, to look at the provenance, the 4.2.0 execution only produced one output node when "stringsAsFactors" was TRUE. Meanwhile, the 3.6.3 execution produced two output nodes, both when the variable's value was and was not specified. Table 3.3 shows the results of this experiment. Note that these executions are from identical scripts using different versions of R.

While this script behaves differently when using different versions of R, it is not especially clear why just by looking at the provenance graphs or examining the code. One may conclude that some component of procedural node 6 is NULL in the first

```
1 printPetNumber <- function(level, df) {
2   number <- nrow(df[df$pet.type==level,])
3   print(paste("There are ", number, " ", level, "s", sep = ""))
4 }
5
6 pet.type <- c("cat", "cat", "dog", "cat", "dog")
7 pet.name <- c("Sterling", "Smuggles", "Snickers", "Katama", "Bode")
8 df3 <- data.frame(pet.name, pet.type)
9 lapply(levels(df3$pet.type), printPetNumber, df3)
```

Figure 3.8: strings-as-factors.R

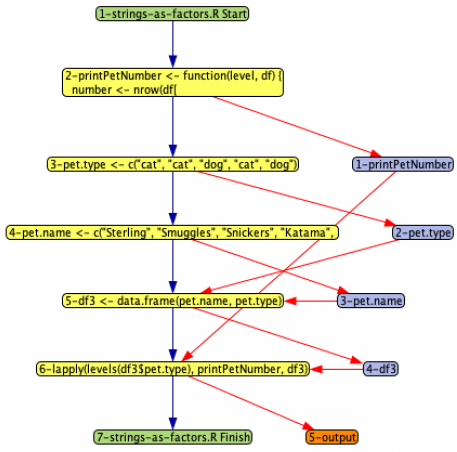
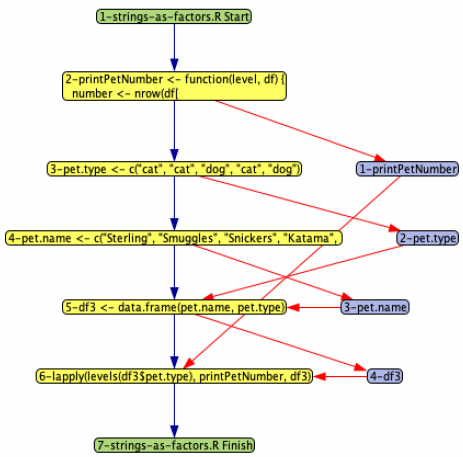
"strings-as-factors.R" in R 3.6.3	"strings-as-factors.R" in R 4.2.0
	
[1] ‘‘There are 3 cats’’ [1] ‘‘There are 2 dogs"	

Table 3.3: Two provenance graphs from the same script, using R 3.6.3 (left) and R 4.2.0 (right)

```
1  x <- c(1, 2, 3, 4, 5, 6, 7, 8)
2
3  if (x > 2){
4    print("it worked!")
5  }
```

Figure 3.9: if-greater-than-one.R

graph; however, nothing explicitly indicates that the difference in output is from language version changes. Therefore, the comparison of the environment, as well as script and input changes, are imperative for deducing why the original results are not reproduced.

Next, I isolated the second significant change to the R language in 4.2.0: using an if-statement to evaluate an expression of length greater than one throws an error. The script, shown in Figure 3.9, declares an array of integers "x" and then uses an if-statement to determine if x was greater than 2. In 4.0.5, the statement returned a warning. However, in 4.2.0, it returns an error. Table 3.4 shows the results from this experiment. While they look identical at first glance, comparing nodes reveals more about the behavioral differences. Note that procedural node 2 outputs an exception node labeled "warning", represented by the red node in R 4.0.5—it outputs an exception node labeled "error" in R 4.2.0.

3.3.3 Final Thoughts From Experimentation

Through this experimentation, I concluded the optimal order for evaluating which changes to the provenance were affecting the behavior of scripts is as follows:

1. Script changes
2. Input file changes
3. Computing environment changes
4. Provenance tool changes

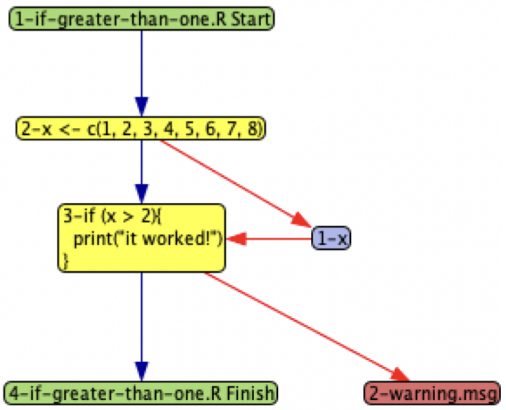
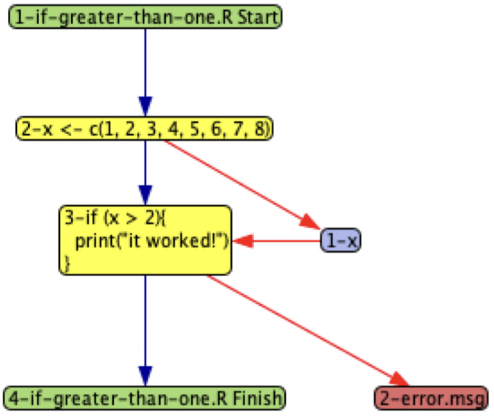
"if-greater-than-1.R" in R 4.0.5	"if-greater-than-1.R" in R 4.2.0
	
node 3 throws a warning message	node 3 throws an error message

Table 3.4: Two provenance graphs from the same script, using R 4.0.5 (left) and R 4.2.0 (right)

If the script itself and its outputs changed, then those script changes are very likely causing the differences in behavior. Script changes are usually intentional; they are meant to change the behavior. If the script has not changed but the input files differ, data nodes created from those input files may also have different values. It is likely the behavioral change coming from the difference in values can be traced back to the difference identified in the input files. Finally, if the script(s) and input files are identical but the behavior still differs, changes to the computing environment remain the most likely cause of different script behavior. If multiple discrepancies in the execution are identified, understanding the effect of script and input changes is typically easier, while understanding how environmental updates affect certain aspects of code behavior is more challenging. By prioritizing the discrepancies which are most likely to substantially affect the script behavior, a user may catch those substantial changes and successfully reproduce results earlier. Therefore, I theorize that it is efficient to address issues in this order when seeking to reproduce a result. In other words, it is more efficient to identify how changes to the computing environment affect script behavior when the environmental discrepancies are isolated from any other script or input changes.

Chapter 4

ADAPTING PROVEXPLAINR

4.1 provExplainR Before Adaptation

4.1.1 Introduciton

As explained in Section 2.3.1, provExplainR is an R package in the End-to-End Provenance suite of packages that compares two prov.JSON files and produces a report of differences between them to help a user identify why the behavior of an execution might be different. It was developed primarily by Mount Holyoke student Khanh Ngo at the Harvard Forest Summer Research Program in Ecology in 2019. Specifically, provExplainR compares the computing environment, library, script(s), provenance tool versions, and input files by using provParseR to extract information from two prov.JSON files a user passes into the "prov.explain()" method. The method looks at each aspect of provenance where differences can occur individually and then reports findings which may be read from the console or saved as a text file.

4.1.2 Limitations

While the original version of provExplainR does a great job identifying script, input, library, and computing environment changes, it does not compare the provenance graphs and their components, such as procedural nodes and data nodes. The prove-

nance graph provides unique information about when the behavior of two executions diverged because it displays explicit connections between data nodes and procedural nodes. Comparing data nodes from the two executions can be especially insightful since they describe variables, errors, warnings, and standard outputs. The original provExplainR does not compare any of these. However, even when the user is aware of data/error discrepancies, knowledge about different values of variables leading up to the production of that error is vital for understanding its origin. There is also the advantage of a more comprehensive report where all the required information for analysis and problem-solving is in one file. Furthermore, provExplainR does not provide the user with any feedback on the interpretation of the differences in provenance and reports on all aspects of the provenance, whether differences occur or not. This makes reading the output of provExplainR difficult, especially for a user with limited knowledge of how these various aspects of provenance may affect their script's behavior. My adaptations to provExplainR seek to address these limitations.

4.2 Approach for provExplainR's Adaptation

My approach for implementation began with developing methods to conduct further comparisons between data nodes and then using the knowledge from my previous experimentation to provide specific feedback to aid a user's attempt to reproduce the results of computational data analysis in R. The code for my version of provExplainR is in Appendix A.

4.2.1 Identifying Changes in Errors and Warnings

My first addition to provExplainR was a function to identify changes in errors and warnings by comparing the exception nodes in the provenance graphs (Figure 4.1), identifying the first difference and reporting the line where the behavior of the two

```
compare.error.nodes <- function(error.report.1, error.report.2, scripts.1, scripts.2){
  cat("\nERROR DIFFERENCE DETECTED:\n")
  diffs.vector[6] <<- TRUE
  if (nrow(error.report.1) >= nrow(error.report.2)){
    end <-  nrow(error.report.1)
    using.1 <- TRUE
  } else {
    end = nrow(error.report.2)
    using.1 <- FALSE
  }
  for (i in 1:end){ #fix for better comparison for when diff lengths
    script.name.1 <- scripts.1[error.report.1[i, "scriptNum"]]
    script.name.2 <- scripts.2[error.report.2[i, "scriptNum"]]
    if(!identical(error.report.1[i, "value"], error.report.2[i, "value"])){
      msg.1 <- error.report.1[i, "value"]
      msg.2 <- error.report.2[i, "value"]
      if(!is.na(error.report.1[i, "startLine"]) && !is.na(error.report.2[i, "startLine"])){
        sl.1 <- error.report.1[i, "startLine"]
        sl.2 <- error.report.2[i, "startLine"]
        if (using.1){
          if(startsWith(msg.1, "Error")){
            cat(paste("Line", sl.1, "of file in Directory 1:", script.name.1, "reported",
                      msg.1, "while in Directory 2:", script.name.2, "reported"))
          } else {
            cat(paste("Line", sl.1, "of file in Directory 1:", script.name.1, "reported Warning",
                      msg.1, "while in Directory 2:", script.name.2, " reported"))
          }
          if (!is.na(msg.2)){
            if(startsWith(msg.2, "Error")){
              cat(paste(msg.2))
            } else {
              cat(paste(" Warning", msg.2))
            }
          } else {
            cat(paste("nothing."))
          }
        } else {
          cat(paste("Line ", sl.2, " of file in Directory 1:", script.name.2, " reported the following: '",
                    msg.2, "' while file in Directory 2:", script.name.1, " reported "))
          if (!is.na(msg.1)){
            cat(msg.2)
          } else {
            cat("nothing")
          }
        }
      }
    }
  }
  cat("\n")
}
```

Figure 4.1: compare.error.nodes() called when a difference in the exception node data frame is detected

executions deviated. provParseR calls nodes representing data, errors, warnings, or standard output as data nodes, so I extracted the data nodes of type "exception" using a function from provParseR. Reusing code from provSummarizeR,[1] a package from the E2E package collection that reports the line an error comes from, I extracted information about each script and found the line number for each exception node. Then, I added each line number to the corresponding row in the data frame. Each row of the data frame also includes which script the line number is from, allowing for cases where an execution involves multiple scripts. The script went row by row, looking for the first error between the exception nodes. Using this approach, I was able to identify the following possible changes: (1) one file produced an error and the other file produced no exceptions, (2) one file produced a warning and the other file produced no exceptions, (3) one file produced an error and the other file produced a different error, (4) one file produced a warning and the other file produced a different warning, and (5) one file produced an error and the other file produced a warning. Often, the execution will halt when an exception occurs, so I focused on identifying the first occurrence of different exception behavior. Also, provParseR returns node information in a data frame, and its comparison relies on the rows of each node corresponding to the same place in the provenance graph. After the first difference, the next set of rows might not refer to the same location on the provenance graph. Once the code encountered a difference, it would report it and state the script, node, and line that produced the difference in behavior. Figure 4.2 shows an example output text block when comparing two provenance files from executions that use expressions of length greater than one in the if statements. For this example, the execution recorded in "Directory 1" used R 4.2 while "Directory 2" used an early version.

[1] `https://github.com/End-to-end-provenance/provSummarizeR`

```
ERROR DIFFERENCE DETECTED:
Line 5 of file in Directory 1: if-greater-than-one.R reported Error in if
(x > 2) {: the condition has length > 1
 while in Directory 2: if-greater-than-one.R reported Warning In  if (x >
2) { :  the condition has length > 1 and only the first element will be
used
```

Figure 4.2: Example output for error comparison with provExplainR

```
DATA NODE DIFFERENCE DETECTED:
d4 from Line 8 in Directory 1 File strings-as-factors-3.R has
different:  valType
compared to d4 from Line 8 in Directory 2 File strings-as-factors-3.R
  d4 valType: {"container":"data_frame", "dimension":[5,2],
"type":["character","character"]}
  d4 valType: {"container":"data_frame", "dimension":[5,2],
"type":["factor","factor"]}
```

Figure 4.3: Example report of differences between data nodes with different value types

4.2.2 Identifying Variable Changes

The code that compared changes in variable values looked at data nodes of type "data" (henceforth referred to as data nodes) and resembled the exception node procedure with a more multifaceted comparison process. This is because there are more qualities of each data node to compare. My method compared each data node's variable name, variable value, and value data type. After constructing data frames with information about data nodes and the lines where they were declared, the function iterates over each row, looking for differences in the three qualities. If the two rows are not identical, the kinds of differences and a line reporting details about each difference are added to the string describing all of the information about the first discrepancy in data nodes. The method can identify all the differences between two data nodes and display them in a readable block of text. Figure 4.3 shows an example report for data node differences in the case where the default value for the stringsAsFactors variable changes depending on the version of R. Therefore, the value type of the data node representing the data frame is different.

```
STDOUT NODE DIFFERENCE DETECTED:
Stdout node d7 of value "There are  3 cat"... in Directory 2 File
strings-as-factors-3.R does not exist in Directory 1
```

Figure 4.4: Example report of differences between standard output node when one execution produces an output and the other does not.

4.2.3 Identifying Changes to Standard Output

The procedure for comparing standard output nodes resembles the previous node comparisons since all three types of nodes are similar. However, standard output nodes have fewer qualities that can vary. Either the message sent to standard output differs, or one script outputs a message while the other does not. Therefore, after constructing analogous data frames with each row corresponding to a standard output node, my code identified the first difference and reported how the standard output behavior diverged. Figure 4.4 shows an example report when a user tries to print the levels of a categorical variable where variable type, either character or factor, is determined by the default value of the stringsAsFactors variable when data.frame() is called. Thus, for one execution, each level name is passed into standard output while the other does not.

4.2.4 Condensing Report on Differences

After adding methods to compare components of the provenance graph, I adapted provExplainR's user report to only display information which describes a change in the provenance. provExplainR originally included messages confirming no changes in an aspect of the provenance if none were found. While this can reassure the user that provExplainR is checking all of the provenance information available, it increases the difficulty of identifying inconsistent components. I modified the methods which inform on the environment, library, input, provenance tool version, and node changes to report only when changes are detected. The technique varied for each method. For environmental differences, I checked the size of the data frame containing all

```
833   # process input files (not URL)
834   input.files.df1 <- input.df1[input.df1$type == "File", ]
835   input.files.df2 <- input.df2[input.df2$type == "File", ]
836
837   empty.1 <- FALSE
838   empty.2 <- FALSE
839   if(nrow(input.files.df1) == 0) {
840     empty.1 <- TRUE
841   }
842   else {
843     get.input.file.changes(input.files.df1, "1")
844   }
845
846   if(nrow(input.files.df2) == 0) {
847     empty.2 <- TRUE
848   }
849   else {
850     get.input.file.changes(input.files.df2, "2")
851   }
852   data.changes <<- FALSE
853   data.not.passed <<- FALSE
854   if(empty.1 && empty.2) return()
855   if(empty.1 && !empty.2){
856     diffs.vector[3] <<- TRUE
857     cat("\nINPUT FILE CHANGES:\n")
858     cat("Directory 2 uses input files while Directory 1 does not.\n")
859     data.not.passed <<- TRUE
860     return()
861   }
862   if (!empty.1 && empty.2){
863     diffs.vector[3] <<- TRUE
864     cat("\nINPUT FILE CHANGES:\n")
865     cat("Directory 1 uses input files while Directory 2 does not.\n")
866     data.not.passed <<- TRUE
867     return()
868   }
869
870
871   # if reached here, both data frames must have some input files
872   # compare files with same name
873   same.name.files.df <- dplyr::inner_join(input.files.df1, input.files.df2, by = "name")
874   compare.input.files.same.name(same.name.files.df)
875
876   # compare files with different name
877   different.name.files.df1 <- dplyr::anti_join(input.files.df1, input.files.df2, by = "name")
878   different.name.files.df2 <- dplyr::anti_join(input.files.df2, input.files.df1, by = "name")
879   compare.input.files.different.name(different.name.files.df1, different.name.files.df2)
880
881 }
```

Figure 4.5: get.input.file.changes() from provExplainR

environmental changes. If it was zero, I exited the function and it did not generate a report. For the provenance tool differences report, I took a virtually identical approach, checking the size of the data frame and storing all changes.

Determining whether or not to report input file differences required a more tailored approach since many different cases ought to prompt the report (Figure 4.5). I created various conditional statements to determine when to print the report header and which messages to include. First, I checked if the executions used input files. If neither did, I exited the function (Figure 4.5, 854). If one did and the other did not,

```r
compare.input.files.same.name <- function (same.name.files.df) {
  if(FALSE == is.null(same.name.files.df) && nrow(same.name.files.df) != 0) {
    # case: same hash value, thus no change detected, no report needed
    same.hash.df <- dplyr::filter(same.name.files.df, same.name.files.df$hash.x == same.name.files.df$hash.y)
    if (nrow(same.hash.df) > 0) {
      # for(i in 1:nrow(same.hash.df)) {
      #   cat("No change detected in the input file", same.hash.df$name[i])
      # }
    }

    # case: different hash value, thus content changed
    different.hash.df <- dplyr::filter(same.name.files.df, same.name.files.df$hash.x != same.name.files.df$hash.y)
    if (nrow(different.hash.df) > 0) {
      if(!diffs.vector[3]){
        cat("\nINPUT FILE CHANGES:\n")
        diffs.vector[3] <<- TRUE
        data.changes <<- FALSE
      }
      for(i in 1:nrow(different.hash.df)) {
        row <- different.hash.df[i, ]
        cat("The content of the input file", row$name, "has changed\n")
        cat("### dir1", row$name, "was last modified at:", row$timestamp.x, "\n")
        cat("### dir2", row$name, "was last modified at:", row$timestamp.y)
      }
    }
  }

}
```

Figure 4.6: compare.input.files.same.name()

```r
compare.input.files.different.name <- function(different.name.files.df1, different.name.files.df2) {
  # case: files with same content but different name, no report needed
  same.hash.df <- dplyr::inner_join(different.name.files.df1, different.name.files.df2, by = "hash")
  if(FALSE == is.null(same.hash.df) && nrow(same.hash.df) != 0) {
    # for(i in 1:nrow(same.hash.df)) {
    #   cat("Content of two input files", same.hash.df$name.x[i], "(dir1) and", same.hash.df$name.y[i], "(dir 2) is the same")
    # }
  }

  # case: files with different hash values and names (dir1)
  exclusive.files1 <- dplyr::anti_join(different.name.files.df1, different.name.files.df2, by = "hash")
  if(FALSE == is.null(exclusive.files1) && nrow(exclusive.files1) != 0) {
    if(!diffs.vector[3]){
      cat("\nINPUT FILE CHANGES:\n")
      diffs.vector[3] <<- TRUE
    }
    cat("\nInput files in dir1 but not in dir2:")
    for(i in 1:nrow(exclusive.files1)) {
      cat("### ", exclusive.files1$name[i], ", which was last modified at ", exclusive.files1$timestamp[i], sep = "")
    }
  }

  # case: files with different hash values and names (dir2)
  exclusive.files2 <- dplyr::anti_join(different.name.files.df2, different.name.files.df1, by = "hash")
  if(FALSE == is.null(exclusive.files2) && nrow(exclusive.files2) != 0) {
    if(!diffs.vector[3]){
      cat("\nINPUT FILE CHANGES:\n")
      diffs.vector[3] <<- TRUE
    }
    cat("\nInput files in dir2 but not in dir1:")
    for(i in 1:nrow(exclusive.files2)) {
      cat("### ", exclusive.files2$name[i], ", which was last modified at ", exclusive.files2$timestamp[i], sep = "")
    }
  }

}
```

Figure 4.7: compare.input.files.different.name()

I reported that difference and exited (Figure 4.5, 855-867). After those conditions were checked, I followed the function execution to points where it created a report of input file differences and printed the header for the report there, but not before checking a boolean variable which told me if I had already printed the header earlier in the method (Figure 4.6, Figure 4.7). Then I would set that boolean variable to TRUE so that if I encountered another difference to report, I would not print the header again (4.6 902-906, 4.7 952-955). Finally, I commented out lines that were for reporting no detected changes (4.6 894-896, 4.7 931-933). The new report would only include information about changes to the input files.

For the functions I created to compare nodes, I compared the data frames containing exception, data, and standard output nodes first. If they were not identical, I called specific functions to conduct a closer comparison of each node type separately. In the specific functions, I started by printing the relevant report header. Thus, if the nodes were identical, the code would not produce a report about them.

I did not omit the report about script differences because the report procedure was more complicated. Instead of printing the report as the function explores different possible inconsistencies, all of the relevant information for each difference is saved to a string variable. Then, the program displays the entire report at the end. There are also a lot more possible differences the code can identify. Differentiating which information is not relevant for the user to know is tricky. Therefore, I opted to keep the original reporting procedure for script changes. It is displayed at the top and minimally crowds the report. It also provides the user with information about which scripts the provenance relates to, which is often referenced in the other parts of the output. A complete report from both the older version and the updated version of provExplainR is available in Appendix B.

```
64  #SCRIPT - 1
65  #LIBRARY - 2
66  #INPUT FILES - 3
67  #ENVIRONMENT - 4
68  #PROV TOOL - 5ip
69  #DATA - 6
70  diffs.vector <<- c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE)
```

Figure 4.8: Declaration of diffs.vector

4.2.5 Summary and Feedback Report

To further increase the readability and usefulness of the provExplainR report, I included a function that summarizes provenance inconsistencies between the two directories and makes suggestions about what may be causing them. The feedback also includes the order to address the causes, informed by my previous experimentation and analysis of how different changes affect provenance.

Creating a Summary

I used a boolean vector to keep track of the type(s) of changes the method identified (Figure 4.8) and constructed a summary of the observed provenance inconsistencies based on which elements in the array were TRUE (Figure 4.9). Each variable in the vector, called "diff.vector", corresponded to a subset of the provenance:

- `diffs.vector[1]`[2] → main script change
- `diffs.vector[2]` → library change
- `diffs.vector[3]` → input files(s) change
- `diffs.vector[4]` → computing environment change
- `diffs.vector[5]` → provenance version tool change
- `diffs.vector[6]` → data/exception/standard out node change

I determined what value to give each vector element by locating points where the code detects the first difference. By default, all elements of diffs.vector are FALSE.

[2]R indexing starts at 1, unlike many other programming languages which start at 0.

```
1387 get.summary <- function(){
1388  if (!any(diffs.vector)){
1389    cat("\nSUMMARY: No differences found.")
1390  } else {
1391    cat("\nSUMMARY: Differences found in the")
1392    if (isTRUE(diffs.vector[1])){
1393      cat("\nscript(s)")
1394    }
1395    if (isTRUE(diffs.vector[2])){
1396      cat("\nlibraries")
1397    }
1398    if (isTRUE(diffs.vector[3])){
1399      cat("\ninput files")
1400    }
1401    if (isTRUE(diffs.vector[4])){
1402      cat("\nenvironment")
1403    }
1404    if (isTRUE(diffs.vector[5])){
1405      cat("\nprovenance tools")
1406    }
1407    if (isTRUE(diffs.vector[6])){
1408      cat("\ndata and/or errors and/or stdout")
1409    }
```

Figure 4.9: Checking contents of diffs.vector in the summary function

```
SUMMARY: Differences found in the
libraries
environment
provenance tools
data and/or errors and/or stdout
```

Figure 4.10: Example summary for two provenance directories where libraries, environments, provenance tool versions and data/errors/stdout have changed.

When the code entered a conditional statement indicating the provenance was different, I set the corresponding element in diffs.vector to TRUE. The assignment could occur multiple times for each element in cases where the program detected multiple differences in one aspect of the provenance. provExplainR records information about each difference in the full report. Thus, details about how many assignments occurred are irrelevant. I used diffs.vector to determine if there were any changes in each aspect of the provenance to determine if the summary should report them. The summary is a list of these differing provenance aspects. Figure 4.10 shows an example summary.

Methodology for Giving Informed Feedback

My main objective throughout this project was to use provenance to aid reproducibility efforts in computational data analysis research. I sought to do this in the provExplainR feedback informed by changes to the provenance. This feedback details how to address inconsistencies in execution behavior. In this section, I will describe what information is in the current version of the feedback as well as my reasoning for including it. In the next and final chapter, I describe the limitations of this approach, solutions addressing those limitations that I did not have time to implement, other methodologies for providing feedback, as well as a summary of the contributions I have made through this research.

Since most of my research involved observing how changes to the R language version affect script execution behavior, I alerted the user to details about significant updates to R's behavior when I determined they could be a cause of behavioral

change. The function comparing computing environments already detects changes to the R language versions. I added conditional statements informed by my previous research to determine if those differences could potentially affect script behavior in critically disruptive ways. For example, if provExplainR found that the computing environment is the most likely cause of change to script behavior and one execution uses R 4.0 or later while the other uses an earlier version, the report includes a specific message alerting the user that these two versions have critical differences that affect script behavior. The feedback also includes a link to the release report for R 4.0, which describes the change in the default value of the stringsAsFactors variable for methods data.frame and read.csv. The program also inserts a similar message if one execution used R 4.2 or later and the other used a version before 4.2. For some of these R version updates, only one critical change in code behavior occurred, such as in the R 4.2 update when only the stringsAsFactors default parameter value changed. However, to make the report more concise and readable, I include a URL to the larger R update report as opposed to reporting details about the change within the provExplainR output. A user may instead read the alert and follow the link to get more information before determining if the update affects their script(s).

In addition to providing insights into how R version change might affect script behavior, the adapted provExplainR advises the user on which provenance differences to address first if attempting to reproduce results from one provenance file. In an intended-use scenario, a user would have an older script and the provenance from a previous execution. They would execute the script on their machine, collect new provenance, and then compare the original and the new provenance. If the script behaves differently, provExplainR can identify differences in the provenance that may be affecting behavior and prioritize which differences to address first. After addressing that difference, they would repeat the process to see if any other script behavior differences persist and address them as well. Determining the order of priority can be very simple, as I have demonstrated. However, there are

opportunities to add complexity, which I will discuss further in the next chapter. My implementation considers which differences are generally most likely to cause behavior differences and which are practical for the user to address. Then, it prioritizes those changes over those which may be affecting behavior in subtler ways and requires more research to resolve. Based on this order, the feedback selects the first change it observes and advises the user to resolve it first, providing resources when available and informing them of where in the full report they can read to find more information.

The order of priority will always start with script changes because if the script(s) change(s), the behavior will almost certainly change. Script changes are also typically the easiest to resolve since provExplainR will pinpoint specific differences based on line number information. If the user finds this report lacking, they can also use the Ubuntu command "diff" to find script changes. Input file changes are second in priority because inputting different data will generally affect script behavior if that script uses that data. Furthermore, errors could occur if a newer execution cannot access the input files that the original could. Like script changes, input file changes are often straightforward to identify since provExplainR will report if the contents within input files change. In cases where the file name is different but the data has not changed, this will not require the user to resolve anything, since behavior will likely not change from differences to the filename. provExplainR accounts for this, only proposing the user address input changes if the contents of the input files are different. However, the detailed report will still record the change in the name of the input files. In addition, if provExplainR finds that one execution used an input file that the other did not, it will inform the user in the feedback, referring them to the more detailed report on input files and recommending they ensure input files are inaccessible directories.

Once a user resolves script and input file changes, differing computing environments and loaded libraries are a likely cause of diverging execution behavior. Often,

the R version affects packages installed by the script, since the install.packages() function determines which packages versions to use based on which version of R is running on the machine. The user can compare all the computing environment changes with the line of code the report identifies as producing a diverging node. From there, the user could parse the provenance to find where functions in that line originate, since the provenance includes information about which package each function comes from, and determine which version updates since the original execution may account for disparate outputs. This could also be facilitated by another program, an opportunity for future development that I will expand on in the next chapter. During this part of the process, it is crucial to consider why a certain update to the behavior of a function, the value of a parameter, or another change, was made. Often, software is updated for good reasons, such as addressing a bug, improving the accuracy of a calculation or even making results more reproducible. Differing results from the new execution may even be more accurate than the original because the code to conduct that analysis has improved. Thus it is always important for the user to think critically about their reproducibility goals when considering how to approach computing environment changes.

In cases where there are no changes to the computing environment that produce significant results but each execution loads different packages, there is a possibility that a function with the same name is producing disparate output. provExplainR will explicitly share if the version of rdtLite—the package used to collect provenance—has changed. By knowing which packages differ between executions and where the function could have come from, the user can determine if the new execution uses a different function with the same name. provExplainR's feedback acts as a guide or map for the user, pointing out specific divergences in the provenance graph and providing likely causes for the user to investigate further. Additionally, provExplainR could be expanded to determine more details about how provenance changes affect external function behavior.

Chapter 5

CONCLUSIONS

My central research question and motivation for development throughout this project concerned how data provenance could aid reproducibility. I first explored avenues for creating computing environments corresponding to the environment described in a provenance file. Then I built a tool, provBuildEnv, which uses Docker to construct these environments. Using these Docker-constructed environments, I executed the same script in different environments, focusing on environmental changes that impacted script execution, to improve the understanding of how different environmental factors can affect the flow of execution displayed in the provenance graph. Finally, I adapted provExplainR to compare the provenance graphs of multiple executions of a script and give informed recommendations for how to resolve reproducibility issues. In this chapter, I will detail the significance of these contributions as well as opportunities for further research.

5.1 Contributions

5.1.1 provBuildEnv

Functionality and Usefulness

provBuildEnv parses a prov.JSON file and constructs a corresponding computing environment using Docker before executing the script in the simulated environment. It is a tool for R that creates a directory with all the files needed to build and run a Docker image with a specified R version, and installs all required packages. A user can opt to either install the package versions specified in the prov.JSON file to reproduce a specific execution, or set the R language version manually and the tool will install the corresponding versions for each package. In addition, the user may edit the requirements.txt file listing all the package dependencies and their versions. This grants the tool some flexibility, as it can be used for both reproducibility when the original prov.JSON file is accessible as well as experimentation when a user wants to understand how R language, package version changes, and inaccessibility or replacement of certain packages might affect the script(s) behavior. Using Docker volumes, users may access all outputs that the execution produces in the Docker container because the 'Data' and 'Analysis' folders inside the container and the Docker image folder on the local machine are shared. In other words, volumes allow data from the local machine to be manipulated by the container and also persist after the container exits. The current version of provBuildEnv's executes the main script with rdtLite's prov.run() and saves the provenance in the 'Data' folder so the user may access data provenance from executions that occur inside the Docker container.

Conclusions from Experimentation

By deploying provBuildEnv's applications for experimentation and supplementary research, two main conclusions can be drawn regarding the interactions between

computing environment variability and execution behavior. First, some R language version changes can prevent computational reproducibility of results. That is, in some cases, running the same script with the same input data can produce different results in different versions of R. Diagnosing that the reproducibility issue relates to version changes in the R language requires an understanding of the versions used for an execution and the significant updates to the language which occurred between them. While R release notes describe all critical updates, most R programmers do not read these carefully or understand their significance until their code stops working, at which point they may find it difficult to track down the source of the problem. Once the changes are known, alterations to the code can resolve most issues.

The second main conclusion is that script and input file changes are far more likely to cause reproducibility issues than environmental ones. This is because most script changes, either intentionally or unintentionally, result in some behavioral change during execution. Furthermore, changes to an input file's data will typically affect the behavior and outputs of a script. Input data file changes are the more likely culprit if all procedural nodes in the provenance graph remain identical, because identical instructions causing different outputs will frequently come from a change in the data before the script ran. In rarer cases where the script relies on the altered mechanics of the R language, the user ought to address environmental disparities. This order also prioritizes which changes are easiest to address since knowledge about the script and input file changes are resolvable with minimal outside research. In contrast, understanding how a version update affects code behavior could require additional research. The updated version of provExplainR uses these conclusions to inform the user about which changes to address first in cases where reproducibility is the goal.

5.1.2 Updated provExplainR

Originally, provExplainR compared data provenance from two executions and reported changes to the scripts, input files, and computing environment. I implemented three major changes that compare components of the provenance graph, omit statements that only verify no differences were detected, and provide feedback and guidance for how to mitigate reproducibility issues.

The updated version of prov.ExplainR provides a more detailed comparison of two provenance files, going beyond the script, input file, and environmental changes to observe how the values of variables, exceptions, and standard output diverged. This additional capability can aid any user seeking to understand how a script's behavior between two executions has changed when attempting to reproduce results. Using this information in tandem with prov.ExplainR's original capacity to compare other provenance changes grants the user a more comprehensive understanding of what makes two executions of similar scripts dissimilar.

By adjusting the provExplainR report to only display information indicating which changes have occurred, the updated version writes a more concise report. Users may quickly and definitively identify inconsistencies between executions using the summary section at the end of the report before seeking details in the text above, allowing for a more direct, piecemeal, and manageable approach to either reproducibility or debugging goals.

provExplainR now also provides feedback guiding users to which aspect of the provenance is the most probable cause of altered script behavior. Gaining insights from experimentation with provBuildEnv directs the user to address detected changes to the script, followed by the input data files, computing environment, and provenance tools. When provExplainR observes multiple changes simultaneously, the program will advise the user to address the most probable change, re-execute the script, recollect provenance, and then conduct a second comparison. In some cases, resolving script and input file changes will eliminate all divergence in script

behavior, even if computing environment differences persist. However, if they do not, addressing computing environment differences may require more critical and context-based decision-making. I will expand on this later.

5.2 Avenues for Future Research

This project demonstrates how provenance can aid in reproducibility efforts for data analysis conducted using the R language and relatively simple scripts and data. In this section, I will propose several opportunities for expansion and recommendations for improving various aspects of the software.

5.2.1 Advocating for Provenance Collection

My work requires a standardized method for using provenance information and uses the End2End provenance package suite to collect provenance and extract data from prov.JSON files. Currently, rdtLite cannot collect provenance for scripts that use raster data, a type of spatial data that uses pixels corresponding with a specific geographic location. rdtLite has other limitations specified in Lerner et al. 2023, which include treating for-loops, if-statements, and function calls as a single unit during execution. This avoids potentially excessive provenance collection time and overall execution time of the scripts, at the cost of losing some more detailed provenance.

Ideally, provenance collection in R would be standardized and commonplace, with tools that allow for levels of flexibility regarding how much detail to collect at the cost of increased overhead. The popularization of standardized provenance collection in the field of science could expand the scope of my work. More provenance collection would provide a drastically larger pool of real-world scripts and data provenance to draw from. Unfortunately, the development of tools that explore the potential of provenance collection and utility is currently stunted by limited transparency and accessibility of real-world scripts, data, and provenance. Policies and

norms which encourage the publishing of data analysis procedures in scientific communities, like those proposed in the 2019 National Academies report, may facilitate this.

5.2.2 Addressing limitations of provBuildEnv

provBuildEnv was not robustly tested on sample scripts besides those created to isolate specific code affected by computing environment changes. To create uncomplicated scenarios for experimentation, each script behaved in specifically distinct ways with different R versions. With more time, provBuildEnv could be tested on other, more complicated, and realistic scripts, such as those collected by Willow Kelleigh in the fall of 2022. However, many of these scripts did not run without errors in the current computing environment. Without data provenance from previous successful executions, these scripts demonstrate reproducibility problems, but are not useful for using provExplainR to resolve those problems. In the best case, having a date associated with the script suggests what R version it used originally. However, complete data provenance provides the most comprehensive list of conditions for successful reproducibility.

There are also opportunities for further testing using artificially created scripts. provBuildEnv has yet to be tested for cases where a main script calls other scripts during execution. I have also not explored in detail how scripts can access inputs through the volumes of the Docker container. If a previous execution used an input file, provBuildEnv makes a copy of that data from the provenance directory and moves it to the "Data" folder within the Docker build directory for easy access through the file system. However, before the Docker image runs, a user must move the input files into the "Analysis" folder to be accessible for the script. Automating this process is tricky because the Docker container has its home directory, and often, scripts contain the pathname of an input file from the local machine. Ensuring the script can find the input file data currently requires some manual tinkering within

the Docker image folder: moving the input files into the same directory as the script(s) and possibly changing the pathname for the input in the script itself. A solution for this would be to parse the prov.JSON file and locate the pathnames for all input files before creating the same set of nested directories besides the script folder. That way, the pathname for each input file is virtually the same inside and outside the container. Recreating the directory system like this may not work in cases where the entire pathname is inside the script.

Another notable limitation of provBuildEnv is that it can only replicate the computing environment from the R language version and libraries. It cannot simulate using a different operating system or hardware. The exploration of potential operating system simulation fell beyond this project's scope. However, operating system differences can affect script behavior since sometimes R depends on built-in operating system functions. If those functions behave differently on various operating systems, the resulting behavior of the script may change.

5.2.3 More User-Friendly Output from provExplainR

The adapted version of provExplainR implements one of many possible methodologies for providing user feedback. Another possible prioritization scheme is to resolve the cause of the first divergence in the provenance graph modeling that script's behavior. This involves comparing all aspects of the provenance graph, not just the data nodes. Instead of looking directly at the script to ascertain if any changes have occurred, the provenance graph would be parsed node by node. If the first difference is a procedural node, provExplainR can conclude that script changes are the most likely cause of the divergence. If the first difference is a data node, including an error, exception, or standard output, provExplainR could trace which procedural node created or altered that data node previously and parse its contexts to identify if the changes may be environmental or related to input data changes.

It is also unclear which method would generate more beneficial feedback for

the user. The current method prioritizes the changes which are more likely to be significant and invites the user to resolve other script and data changes before investigating whether addressing computing environment changes makes sense in that scenario. Meanwhile, parsing the provenance graph and addressing the first divergence might allow the researcher to reproduce results more efficiently. More experimentation is necessary to determine if the first behavioral divergence often produces ripple effects, whereby one change could produce a more broadly significant change to the script behavior. If this is found to be true, addressing the first point in the provenance graph ought to be prioritized, since its resolution could reveal more differences that were previously undetected.

There are also opportunities for further extending the capabilities of provExplainR to aid reproducibility efforts. Currently, provExplainR encourages user-led investigation into the relationship between expressions that produce diverging outputs and the components of the computing environment that affect pieces of that expression, such as packages and language version. However, since the prov.JSON file contains information about the origins of external functions and data, it is feasible for provExplainR to give more specific feedback about which package changes might impacting reproducibility. provExplainR could even be maintained to recognize common reproducibility issues in a variety of popular packages. Testing and follow-up on a large collection of sample scripts could reveal which specific provenance updates are most common. By creating a dynamically updated database of these critical version updates, provExplainR could query the relevant package version updates based on which external functions are called in the expressions producing divergent outputs.

5.3 Final Thoughts

This thesis involved exploratory experimentation and development to better understand data provenance's capability to aid reproducibility efforts. I conclude that access to data provenance from previous executions, especially when stored in a standardized and parseable file format, could help address the modern reproducibility crisis in computational data analysis research. However, depending on the user's goals, resolving computing environment differences may not be the best idea in cases where the updated version produces different but more accurate results. Understanding the purpose of a software update may suggest prioritizing computational reproducibility is misguided and instead provide an opportunity to re-evaluate the accuracy of the original results. In many cases, script maintenance, whereby a developer or researcher makes deliberate changes to a script in order to improve functionality, accuracy of the results, or incorporate a new capability, should be prioritized over exact reproduction of the original results. However, reproducibility should remain a first step, cultivating a deeper understanding of how the original data analysis methods may need to change to accommodate updated software. Regardless, movements to popularize and eventually expect the open access of scripts, data, and data provenance working presently with the development of tools that use provenance like provBuildEnv and provExplainR could remediate the reproducibility crisis and improve our abilities to verify scientific results.

Appendix A

provBuildEnv

```
    prov.BuildEnv <- function(prov.dir, script.name, docker.
        image.name, from.prov.file = TRUE, r.version = NULL){
  if (!file.exists(prov.dir)) {
    stop("Provenance directory not found")
  }
  prov.file <- paste(prov.dir, "/prov.json", sep= "")

  if (!file.exists(prov.file)) {
    stop("Provenance file not found")
  }

  prov <- provParseR::prov.parse(prov.file)

  build.env.dir(prov, docker.image.name, prov.dir, script.
      name, from.prov.file, r.version)
}

build.env.dir <- function(prov, docker.image.name, prov.dir,
    script.name, from.prov.file, r.version){
  build.dir<- paste(prov.dir, "/", docker.image.name, sep
      ="")
  if (!file.exists(build.dir)){
    dir.create(build.dir)
  }
  docker.file <- paste(build.dir, "/Dockerfile", sep="")
```

```
if (!file.exists(docker.file)){
  file.create(docker.file)
}
entrypoint <- paste(build.dir, "/entrypoint.sh", sep="")
if (!file.exists(entrypoint)){
  file.create(entrypoint)
}
docker.compose <- paste(build.dir, "/docker-compose.yml",
   sep="")
if (!file.exists(docker.compose)){
  file.create(docker.compose)
}
volumes.dir <- paste(build.dir, "/volumes", sep="")
if (!file.exists(volumes.dir)){
  dir.create(volumes.dir)
}
requirements <- paste(volumes.dir, "/requirements.txt",
   sep="")
if (!file.exists(requirements)){
  file.create(requirements)
}
analysis.dir <- paste(volumes.dir, "/Analysis/", sep = "")
if (!file.exists(analysis.dir)){
  dir.create(analysis.dir)
}
data.dir <- paste(volumes.dir, "/Data/", sep = "")
if (!file.exists(data.dir)){
  dir.create(data.dir)
```

```
}

environment <- provParseR::get.environment(prov)
if(is.null(r.version)){
  r.version <- environment[environment$label == "
     langVersion", ]$value
  r.version <- get.version.type(r.version)
}



script.path <- paste(prov.dir, "/scripts/", sep = "")
data.path <- paste(prov.dir, "/data/", sep = "")

script.files <- list.files(script.path)
data.files <- list.files(data.path)

file.copy(from = paste0(script.path, script.files), to =
   paste0(analysis.dir, script.files))
file.copy(from = paste0(data.path, data.files), to =
   paste0(data.dir, data.files))

#write dockerfile
writeLines(c(paste("FROM rocker/r-ver:", r.version, sep
   =""),
             "LABEL Maintainer=\"sfabrega\"",
             "WORKDIR /home",
             "COPY entrypoint.sh /entrypoint.sh",
             "RUN chmod 755 /entrypoint.sh",
```

```
                "ENTRYPOINT [\"/entrypoint.sh\"]"
), docker.file)

libs <- provParseR::get.libs(prov)
script.libraries <- libs[libs$whereLoaded == "script", ] #
   loaded because script used them
preloaded.libraries <- libs[libs$whereLoaded == "preloaded
   ", ] #libs loaded when starts
rdtLite.libraries <- libs[libs$whereLoaded == "rdtLite", ]
    #loaded because rdtLite uses them, possible that
   script uses them
unknown.libraries <- libs[libs$whereLoaded == "unknown", ]
    #loading in a prov file if old

script.vals <- data.frame(script.libraries$name, script.
   libraries$version)
preloaded.vals <- data.frame(preloaded.libraries$name,
   preloaded.libraries$version)
rdtLite.vals <- data.frame(rdtLite.libraries$name, rdtLite
   .libraries$version)
unknown.vals <- data.frame(unknown.libraries$name, unknown
   .libraries$version)


lines <- set.requirements.lines(script.vals, preloaded.
   vals, rdtLite.vals, unknown.vals)
writeLines(as.character(lines), requirements)
```

```
writeLines(c("version: '1'",
             "services:",
             "  rscript:",
             paste("    image: ",docker.image.name, sep
                =""),
             "    volumes:",
             paste("      - ", volumes.dir, ":/home", sep
                =""),
             "    environment:",
             "      - TZ=America/New_York",
             paste("    command: Rscript -e \"rdtLite::
                prov.run(\'Analysis/", script.name,"\',
                prov.dir = \'Data\')\"",sep="")),
           docker.compose)

if (from.prov.file){
  writeLines(c("#!/bin/bash",
               "cd /home",
               "apt update",
               "DEBIAN_FRONTEND=noninteractive apt-get --
                  yes install build-essential libcurl4-
                  gnutls-dev libxml2-dev",
               "FILE=./requirements.txt",
               "if test -f \"$FILE\"; then",
               " echo \"$FILE exists.\" ",
               " Rscript -e \"install.packages('remotes')
                  \"",
               " while read -r package version; ",
```

```
                " do ",
                "  Rscript -e \"remotes::install_version
                   ('\"$package\"', version='\"$version\"',
                    repos = 'https://cran.rstudio.com/')\";
                    ",
                " done < \"$FILE\"",
                "fi",
                " Rscript -e \"install.packages('rdtLite')
                   \"",
                "exec \"$@\" # run whatever command is
                   given for "command:" in docker-compose.
                   yml"),
              entrypoint)
}else {
  writeLines(c("#!/bin/bash",
                "cd /home",
                "apt update",
                "DEBIAN_FRONTEND=noninteractive apt-get --
                   yes install build-essential libcurl4-
                   gnutls-dev libxml2-dev",
                "FILE=./requirements.txt",
                "PACKAGE_LS=\"c(\"",
                "if test -f \"$FILE\"; then",
                " echo \"$FILE exists.\" ",
                " while read -r package version; ",
                " do ",
                "  PACKAGE_LS+=\"\'${package}\',\"",
                " done < \"$FILE\"",
```

```
                    " PACKAGE_LS=${PACKAGE_LS::-1}",
                    " PACKAGE_LS+=\",'rdtLite')\"",
                    " echo ${PACKAGE_LS} ",
                    "fi",
                    " Rscript -e \"install.packages(${
                       PACKAGE_LS})\"",
                    "exec \"$@\" # run whatever command is
                       given for "command:" in docker-compose.
                       yml"),
                  entrypoint)

  }




}
set.requirements.lines <- function(script.df, preloaded.df,
   rdtLite.df, unknown.df){
  lines <- c("#package requirements")
  script.names <- script.df$script.libraries.name
  script.versions <- script.df$script.libraries.version
  for (i in seq_len(length(script.names))){
    if (!is.base.package(script.names[i])){
      lines <- c(lines, paste(script.names[i], script.
         versions[i], sep=" "))
    }
  }
```

```
  preloaded.names <- preloaded.df$preloaded.libraries.name
  preloaded.versions <- preloaded.df$preloaded.libraries.
     version
  for (i in seq_len(length(preloaded.names))){
    if (!is.base.package(preloaded.names[i])){
      lines <- c(lines, paste(preloaded.names[i], preloaded.
         versions[i], sep=" "))
    }
  }
  rdtLite.names <- rdtLite.df$rdtLite.libraries.name
  rdtLite.versions <- rdtLite.df$rdtLite.libraries.version
  for (i in seq_len(length(rdtLite.names))){
    if (!is.base.package(rdtLite.names[i])){
      lines <- c(lines, paste(rdtLite.names[i], rdtLite.
         versions[i], sep=" "))
    }
  }
  unknown.names <- unknown.df$unknown.libraries.name
  unknown.versions <- unknown.df$unknown.libraries.version
  for (i in seq_len(length(unknown.names))){
    if (!is.base.package(unknown.names[i])){
      lines <- c(lines, paste(unknown.names[i], unknown.
         versions[i], sep=" "))
    }
  }
  return(lines)

}
```

```
get.version.type <- function(r.version){
  r.version <- unlist(strsplit(r.version, " "))[3]
  return(r.version)
}
is.base.package <- function(name){
  base.packages = c('base', 'compiler', 'datasets',
                    'graphics', 'grDevices', 'grid',
                    'methods', 'parallel', 'splines',
                    'stats', 'stats4', 'tcltk',
                    'tools', 'utils')
  if(name %in% base.packages){
    return(TRUE)
  }
  return(FALSE)
}
```

Appendix B

Older provExplainR sample output

```
You entered:
dir1 = /Users/seanfabrega/Desktop/workspace/
   scripts_for_ce_testing/prov_testing/prov_strings-as-
   factors-3
dir2 = /Users/seanfabrega/Desktop/workspace/
   scripts_for_ce_testing/prov_testing/prov_strings-as-
   factors-3/saf-3-363/volumes/Data/prov_strings-as-factors
   -3
```

```
SCRIPT CHANGES: The content of the main script strings-as-
    factors-3.R has changed
Run prov.diff.script to see the changes.
### dir1 main script strings-as-factors-3.R was last
    modified at: 2023-04-03T21.12.19EDT
### dir2 main script strings-as-factors-3.R was last
    modified at: 2023-04-03T21.15.14EDT

LIBRARY CHANGES:
Library version differences:
      name dir1.version dir2.version
      base         4.2.0         3.6.3
  datasets         4.2.0         3.6.3
   ggplot2         3.4.0         3.3.0
  graphics         4.2.0         3.6.3
 grDevices         4.2.0         3.6.3
   methods         4.2.0         3.6.3
     stats         4.2.0         3.6.3
     utils         4.2.0         3.6.3

Libraries in dir2 but not in dir1:
No such libraries were found
Libraries in dir1 but not in dir2:        name version
  assertthat    0.2.1
         bit    4.0.4
       bit64    4.0.5
         cli    3.6.0
  colorspace    2.0-3
```

compiler	4.2.0
crayon	1.5.1
DBI	1.1.2
digest	0.6.29
dplyr	1.0.9
ellipsis	0.3.2
fansi	1.0.3
generics	0.1.2
glue	1.6.2
grid	4.2.0
gtable	0.3.0
jsonlite	1.8.0
lifecycle	1.0.3
magrittr	2.0.3
munsell	0.5.0
pillar	1.7.0
pkgconfig	2.0.3
purrr	0.3.4
R6	2.5.1
rdtLite	1.4
rlang	1.0.6
rstudioapi	0.13
scales	1.2.0
sessioninfo	1.2.2
stringi	1.7.6
tibble	3.1.7
tidyselect	1.1.2
tools	4.2.0

```
 tzdb    0.3.0
 utf8    1.2.2
vctrs    0.5.2
vroom    1.5.7
```

```
INPUT FILE CHANGES:
No input files were found in dir 1
No input files were found in dir 2

ENVIRONMENT CHANGES: Value differences:
 Attribute: architecture
### dir1 value: x86_64, darwin17.0
### dir2 value: x86_64
 Attribute: operating system
### dir1 value: macOS Monterey 12.4
### dir2 value: linux-gnu
 Attribute: language version
### dir1 value: R version 4.2.0 (2022-04-22)
### dir2 value: R version 3.6.3 (2020-02-29)
 Attribute: total elapsed time
### dir1 value: 0.925
### dir2 value: 5.724
 Attribute: working directory
### dir1 value: /Users/seanfabrega/Downloads
### dir2 value: /home
 Attribute: provenance directory
### dir1 value: /Users/seanfabrega/Desktop/workspace/
```

```
    scripts_for_ce_testing/prov_testing/prov_strings-as-
    factors-3
### dir2 value: Data/prov_strings-as-factors-3
 Attribute: provenance collection time
### dir1 value: 2023-04-03T21.13.21EDT
### dir2 value: 2023-04-18T15.34.40EDT
 Attributes in dir1 but not in dir2:
       label                                        value
          ui 2022.02.2+485 Prairie Trillium (desktop)
      pandoc                                          NA
  scriptHash          1f8534dfdca811649c27cc3587b25a54

PROVENANCE TOOL CHANGES: Tool differences: Name: rdtLite###
    dir1 tool version: 1.4 ; dir1 json version: 2.3### dir2
    tool version: 1.2.1 ; dir2 json version: 2.3
```

Adapted provExplainR sample output

```
You entered:
dir1 = /Users/seanfabrega/Desktop/workspace/
    scripts_for_ce_testing/prov_testing/prov_strings-as-
    factors-3
dir2 = /Users/seanfabrega/Desktop/workspace/
    scripts_for_ce_testing/prov_testing/prov_strings-as-
    factors-3/saf-3-363/volumes/Data/prov_strings-as-factors
    -3
SCRIPT CHANGES:
No change detected in the content of the main script strings
```

```
  -as-factors-3.R
### dir1 main script strings-as-factors-3.R was last
  modified at: 2023-04-03T21.12.19EDT
### dir2 main script strings-as-factors-3.R was last
  modified at: 2023-04-03T21.15.14EDT
LIBRARY DIFFERENCES DETECTED:
      name dir1.version dir2.version
      base         4.2.0         3.6.3
  datasets         4.2.0         3.6.3
   ggplot2         3.4.0         3.3.0
  graphics         4.2.0         3.6.3
 grDevices         4.2.0         3.6.3
   methods         4.2.0         3.6.3
     stats         4.2.0         3.6.3
     utils         4.2.0         3.6.3

Libraries in dir1 but not in dir2:       name version
  assertthat    0.2.1
         bit    4.0.4
       bit64    4.0.5
         cli    3.6.0
  colorspace    2.0-3
    compiler    4.2.0
      crayon    1.5.1
         DBI    1.1.2
      digest   0.6.29
       dplyr    1.0.9
    ellipsis    0.3.2
```

fansi	1.0.3
generics	0.1.2
glue	1.6.2
grid	4.2.0
gtable	0.3.0
jsonlite	1.8.0
lifecycle	1.0.3
magrittr	2.0.3
munsell	0.5.0
pillar	1.7.0
pkgconfig	2.0.3
purrr	0.3.4
R6	2.5.1
rdtLite	1.4
rlang	1.0.6
rstudioapi	0.13
scales	1.2.0
sessioninfo	1.2.2
stringi	1.7.6
tibble	3.1.7
tidyselect	1.1.2
tools	4.2.0
tzdb	0.3.0
utf8	1.2.2
vctrs	0.5.2
vroom	1.5.7

ENVIRONMENT DIFFERENCES DETECTED:

```
Attribute: architecture
### dir1 value: x86_64, darwin17.0
### dir2 value: x86_64
Attribute: operating system
### dir1 value: macOS Monterey 12.4
### dir2 value: linux-gnu
Attribute: language version
### dir1 value: R version 4.2.0 (2022-04-22)
### dir2 value: R version 3.6.3 (2020-02-29)
Attribute: total elapsed time
### dir1 value: 0.925
### dir2 value: 5.355
Attribute: working directory
### dir1 value: /Users/seanfabrega/Downloads
### dir2 value: /home
Attribute: provenance directory
### dir1 value: /Users/seanfabrega/Desktop/workspace/
    scripts_for_ce_testing/prov_testing/prov_strings-as-
    factors-3
### dir2 value: Data/prov_strings-as-factors-3
Attribute: provenance collection time
### dir1 value: 2023-04-03T21.13.21EDT
### dir2 value: 2023-04-03T22.22.28EDT
Attributes in dir1 but not in dir2:
      label                                    value
         ui 2022.02.2+485 Prairie Trillium (desktop)
     pandoc                                       NA
 scriptHash          1f8534dfdca811649c27cc3587b25a54
```

```
PROVENANCE TOOL DIFFERENCES DETECTED:
Name: rdtLite### dir1 tool version: 1.4 ; dir1 json version:
    2.3### dir2 tool version: 1.2.1 ; dir2 json version: 2.3
DATA NODE DIFFERENCE DETECTED:
d4 from Line 8 in Directory 1 File strings-as-factors-3.R
   has different:  valType
compared to d4 from Line 8 in Directory 2 File strings-as-
   factors-3.R
  d4 valType: {"container":"data_frame", "dimension":[5,2],
     "type":["character","character"]}
  d4 valType: {"container":"data_frame", "dimension":[5,2],
     "type":["factor","factor"]}

STDOUT NODE DIFFERENCE DETECTED:
Stdout node d7 of value "There are  3 cat"... in Directory 2
    File strings-as-factors-3.R does not exist in Directory
   1

SUMMARY: Differences found in the
libraries
environment
provenance tools
data and/or errors and/or stdout
FEEDBACK:
Differences are likely from environment changes. Review how
   the computing environment changes affected script lines
   referenced in the data/error node output.
```

Different R versions detected:

R 3.x and 4.x have critical differences that affect script behavior. See https://stat.ethz.ch/pipermail/r-announce/2020/000653.html for more information.

R 4.2.x has critical differences from previous version that affect script behavior. See https://stat.ethz.ch/pipermail/r-announce/2022/000683.html for more information.

Bibliography

[1] *Reproducibility and Replicability in Science*. National Academies Press, September 2019.

[2] Adler, J. *R in a Nutshell*, 2nd ed. O'Reilly, 2012.

[3] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* (2004), pp. 423–424.

[4] Beaulieu-Jones, B. K., and Greene, C. S. Reproducibility of computational workflows is automated using continuous analysis. *Nature Biotechnology 35*, 4 (March 2017), 342–346.

[5] Boettiger, C. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review 49*, 1 (jan 2015), 71–79.

[6] Boettiger, C., and Eddelbuettel, D. An Introduction to Rocker: Docker Containers for R. *The R Journal 9*, 2 (2017), 527.

[7] Boose, E. R., Lerner, B. S., and KRESS, W. J. *Replication of Data Analyses: PROVENANCE IN R.* Yale University Press, 2017, pp. 195–212.

[8] Chambers, J. M. S, R, and Data Science. *The R Journal 12*, 1 (2020), 462.

[9] Cheney, J., Ahmed, A., and Acar, U. A. Provenance as dependency analysis. *Mathematical Structures in Computer Science 21*, 6 (oct 2011), 1301–1337.

[10] Clark, D., Culich, A., Hamlin, B., and Lovett, R. Bce: Berkeley's common scientific compute environment for research and education.

[11] Collberg, C., Proebsting, T., Moraila, G., Shankaran, A., Shi, Z., and Warren, A. M. Measuring reproducibility in computer systems research.

[12] Davidson, S. B., and Freire, J. Provenance and scientific workflows: Challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, Association for Computing Machinery, pp. 1345–1350.

[13] Dudley, J. T., and Butte, A. J. In silico research in the era of cloud computing. *Nature Biotechnology 28*, 11 (nov 2010), 1181–1185.

[14] Eddelbuettel, D., and François, R. Rccp: Seamless R and C++ Integration. *Journal of Statistical Software 40*, 8 (2011).

[15] Giorgi, F. M., Ceraolo, C., and Mercatelli, D. The R Language: An Engine for Bioinformatics and Data Science. *Life 12*, 5 (April 2022), 648.

[16] Howe, B. *Implementing Reproducible Research*, 1st ed. Chapman and Hall/CRC, 2014, ch. Reproducibility, Virtual Appliances, and Cloud Computing.

[17] Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M. R., Li, P., and Oinn, T. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research 34*, Web Server (July 2006), 729–732.

[18] Huynh, T. D., Jewell, M. O., Keshavarz, A. S., Michaelides, D. T., Yang, H., and Moreau, L. The prov-json serialization. W3C Member Submission, April 2013.

[19] Ihaka, R., and Gentleman, R. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics 5*, 3 (sep 1996), 299.

[20] Ince, D. C., Hatton, L., and Graham-Cumming, J. The case for open computer programs. *Nature 482*, 7386 (February 2012), 485–488.

[21] Kalibera, T., Meyer, S., and Hornik, K. Changes in R 3.6–4.0. *The R Journal 12/2* (December 2020).

[22] Kalibera, T., Meyer, S., and Hornik, K. The R Journal: Changes in R. *The R Journal 14* (2022), 361–364. https://journal.r-project.org/news/RJ-2022-4-rcore.

[23] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* (September 2017), 37 Pages.

[24] Lerner, B., Boose, E., Brand, O., Ellison, A. M., Fong, E., Lau, M., Ngo, K., Pasquier, T., Perez, L. A., Seltzer, M., Sheehan, R., and Wonsil, J. Making provenance work for you. *The R Journal 14*, 4 (feb 2023), 141–159.

[25] Monniaux, D. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems 30*, 3 (May 2008), 1–41.

[26] Nüst, D., Eddelbuettel, D., Bennett, D., Cannoodt, R., Clark, D., Daróczi, G., Edmondson, M., Fay, C., Hughes, E., Kjeldgaard, L., Lopp, S., Marwick, B., Nolis, H., Nolis, J., Ooi, H., Ram, K., Ross, N., Shepherd, L., Sólymos, P., Tyson, Swetnam, L., Turaga, N., Petegem, C. V., Williams, J., Willis, C., and Xiao, N. The Rockerverse:

Packages and Applications for Containerisation with R. *The R Journal 12*, 1 (2020), 437.

[27] Silles, C. A., and Runnalls, A. R. Provenance-Awareness in R. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 64–72.

[28] Tippmann, S. Programming tools: Adventures with R. *Nature 517*, 7532 (dec 2014), 109–110.

[29] Wonsil, J. *Reproducibility as a service*. PhD thesis, University of British Columbia, 2021.