Containerized Analyses Enable Interactive and Reproducible Statistics

Gregory J. Hunt
Department of Mathematics, William & Mary
and
Johann A. Gagnon-Bartsch
Department of Statistics, University of Michigan

March 3, 2021

Abstract

In recent decades the analysis of data has become increasingly computational. Correspondingly, this has changed how scientific and statistical work is shared. For example, it is now commonplace for underlying analysis code and data to be proffered alongside journal publications and conference talks. Unfortunately, sharing code faces several challenges. First, it is often difficult to take code from one computer and run it on another. Code configuration, version, and dependency issues often make this challenging. Secondly, even if the code runs, it is often hard to understand or interact with the analysis. This makes it difficult to assess the code and its findings, for example, in a peer review process. In this paper we advocate for two practical approaches to help make sharing interactive and reproducible analyses easy: (1) analysis containerization, a technology that fully encapsulates an analysis, data, code and dependencies into a shareable format, and (2) code notebooks, an accessible format for interacting with third-party analyses. We will demonstrate that the combination of these two technologies is powerful and that containerizing interactive code notebooks can help make it easy for statisticians to share code, analyses, and ideas.

Keywords: containerization, code notebooks, results caching

1 Introduction

Before the widespread adoption of peer-reviewed scientific journals, it was not uncommon for scientists to keep their findings secret. Famously, Leonardo Da Vinci wrote in mirrored handwriting to obfuscate his notebooks and Isaac Newton kept hidden his development of calculus for nearly forty years (National Academy of Sciences et al., 2009). Modern science, however, advances through a rich process of open and timely sharing. Today, there are a plethora of ways to share results such as talks at conferences, proceedings, seminars, posters, peer-reviewed literature and pre-print repositories. Open sharing not only allows results to be disseminated and built upon, but also allows scrutiny and verification of the research and is fundamental to the scientific process itself. However, as scientific analysis has progressed, so too has the notion of sharing. In particular, the last several decades have seen the analysis of scientific data become heavily computational. This is especially true of statistical work, where coding has become deeply intertwined with statistical analysis. Correspondingly, the notion of what it means to share research results has also expanded (Ellis and Leek, 2018). The modern notion of sharing research encompasses not only sharing prose and proofs, but also sharing code and data.

It is now commonplace for data and the accompanying analysis code to be shared through online repositories. Indeed, many peer-reviewed journals require it. For general purpose code, a popular sharing platform is github (Github, Inc., 2021). Language specific repositories for software packages also exist, e.g. CRAN for R packages (The R Project for Statistical Computing, nd) or PyPI for Python (Python Software Foundation, 2021). Moderately sized datasets may also be hosted on github or Kaggle (Kaggle Inc., 2019). This open sharing of analysis code is a growing trend in statistics. Nonetheless, it faces several practical challenges. Two important challenges are (1) actually running the shared code, and (2) understanding and interacting with the code.

The first challenge is that code that runs on one computer may not always run on another. For example, the package may not be available for the current version of the language or dependencies of the package may fail to install. Modern analysis often relies on a large and complex collection of interdependent software packages and thus there are many places for such version or dependency issues to arise. Similarly, directory structures across

machines may not be identical and, for example, data, code, or other files may not reside where the analysis is expecting. Fixing such problems often entails a significant investment of time and energy. For example, troubleshooting failed installations of dependencies can often lead down a chain of fixing cryptic installation errors which is difficult even for an experienced user.

In addition to the challenges of taking analysis from one computer and running it on another, a second major challenge is difficulty understanding or interacting with code. While it may be impractical or unnecessary to insist on understanding code on a line-by-line basis, a lot can be learned about an analysis by making small modifications to code. For example, one can explore different parameter settings or function arguments and see how output changes. Here, simply sharing raw code is often inadequate. Moderately complicated code, even well-written code, can be difficult to understand and explore. Consequently, it is often difficult for third-parties to find reasonable entry-points into code to modify or scrutinize the analysis.

These issues with running, understanding, and interacting with code inhibit the sharing of statistical and scientific results. To help overcome these issues, this paper will advocate for two practical strategies: (1) analysis containerization, and (2) interactive notebooks. Containerization is a technology that allows encapsulation of an entire system including data, code, dependencies, and programs into a reproducible, shareable, and self-contained format. When a third party takes the container and runs it on their own computer, it will be as if they are instead working in the computational environment where the analysis was originally done. All of the programs, files, code, data, and configurations will be exactly reproduced as they were in that original environment. While any format or organization of analysis code can be containerized, we advocate in particular for containerizing interactive code notebooks. Notebooks are an increasingly popular document format that allow natural interweaving of commentary, code, and output. We believe that these notebooks make for some of the most clear, concise, and intuitive ways of documenting and interacting with the code and analysis.

This paper will use several free and open-source tools to provide concrete examples of the proposed strategies. In particular, we will use Docker (Docker Inc., 2021b) for con-

tainerization and jupyter with jupytext as an environment for writing and interacting with code notebooks (Project Jupyter, 2021). However, the general containerization approach we advocate is not tied to any particular choices of technology.

The remainder of this paper is organized as follows. Section 2 introduces containerization. Section 3 discusses interactive notebooks and shows how they can be effectively containerized. Section 4 concludes with a discussion of reproducible sharing and beyond.

2 Containerizing Analyses

Containerization solves many of the dependency and configuration issues that otherwise make sharing truly reproducible analyses difficult. Figure 1 displays a high-level overview of sharing of a containerized analysis. First, the entire computing environment in which the analysis was originally run is encapsulated into a single file. This file, called an image, is essentially a copy of the system on which the analysis was conducted. The image file may be shared, for example, by uploading it to the cloud. From there, the image may be downloaded by a third party and, with just a few keystrokes, the third party is placed into a duplicate of the original computing environment. All of the data, code, dependencies, configurations and software are precisely set up as in the original environment, and thus set up to reproduce the analysis exactly. The goal of containerization is to ensure that if the code worked when containerized, it will work when the image is run by a third party. Containerizing and sharing analyses is a simple process akin to uploading code to github. However, unlike uploading code to github, containerizing analyses ensures exact computational reproducibility and, as we will show, enables natural interaction with the shared analyses.

Containerization works by quarantining off a portion of a host computer's resources and allocating them to a virtual operating system (OS) that lives within the host OS. This virtual OS is spawned from the image file, and a single image may be used to spawn multiple instances of the virtual OS. These instances are called containers, hence the term containerization. An "image" refers to the actual file that may be uploaded, downloaded or shared, while a "container" refers to an ephemeral instance running on the computer. While virtualization isn't new, containerization is the latest incarnation of the technology

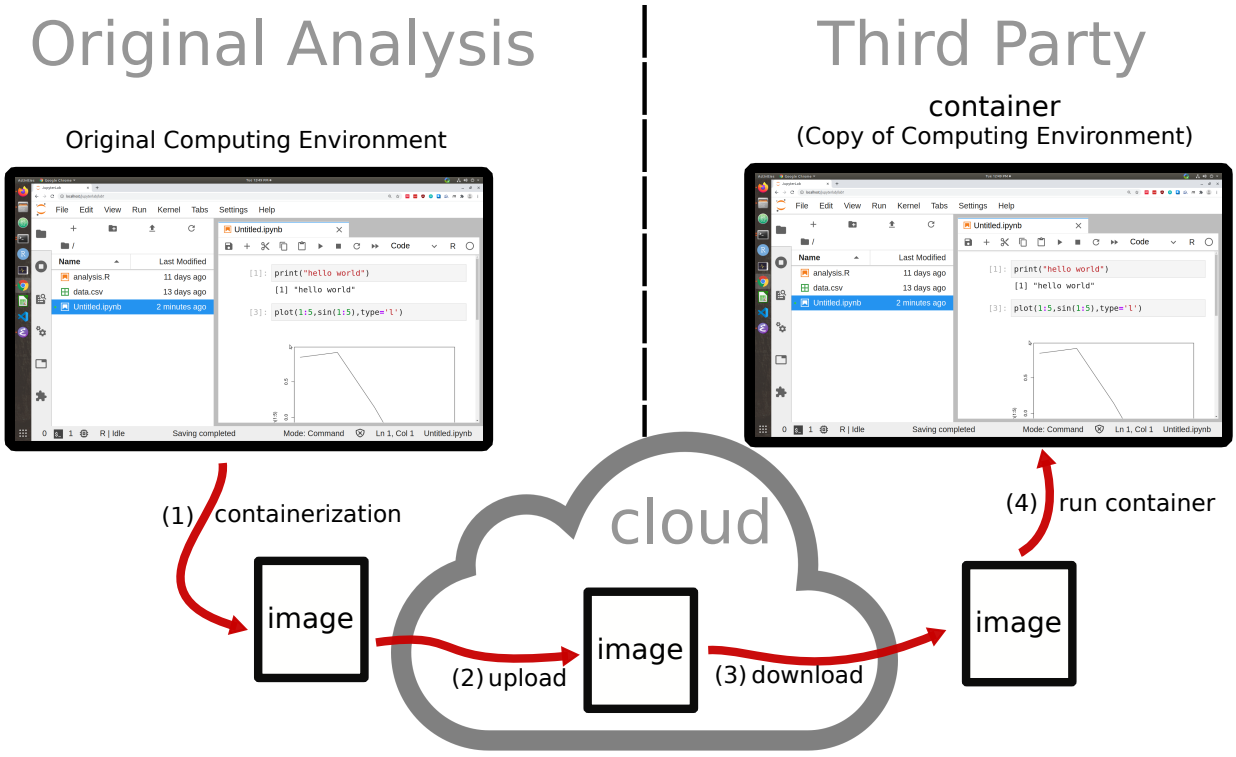

Figure 1: Typical sharing of containerized analysis. (1) The computing environment is containerized, creating a self-contained image file. (2) This image file may be uploaded to the cloud and then (3) downloaded by a third party. (4) From there, the third party may use the image to re-create the original computing environment.

and comes with several key advantages over its predecessors. Primarily, containerization is incredibly light-weight. Containers only virtualize the high-level components of the OS (e.g. code, configuration files, software and data) and seamlessly re-use the low-level processing components of the host OS. Indeed, starting up a container doesn't actually start up a second instance of an OS; it largely just changes all references for resources, system libraries, files, and data, to refer to a particular isolated section of the computer. The light-weight nature of such containers means that the images are small on disk and hence portable, making them quick to upload, download, and share. Furthermore, since starting a container largely just changes the references to resources in the environment, containers start up nearly instantaneously and run code at speeds nearly identical to the host OS (Felter et al., 2015).

2.1 A Practical Example

To give a feel for the simplicity of containerization, Figure 2 displays an example workflow using Docker.[1] Upon starting the container, the user is immediately placed into a new computing environment. We have set up this environment with a pre-loaded data file, an R script reproducing analysis of the data, and an installation of R. In addition to merely allowing inspection of the data or scripts, containerization lets the user actually run the code in the pre-configured environment. Running the script uses the version of R installed in the container. We have set up this version of R with all of the necessary packages and configurations to run the analysis. Thus the code is guaranteed to run without further tinkering or troubleshooting. In our example, the code produces a plot and saves it as a PDF. After conducting the analysis, one can leave the container's environment and jump back into the host computer's environment. Leaving the container does not delete it, but merely pauses it. Consequently, since the container still exists in the background, one can copy out any of its contents. In this example we extract the newly created PDF. Conversely, files may also be transferred in the other direction, from the host to the container, in a similar fashion. Finally, after we are completely done using the container, we remove it so

[1] Runnable versions of the examples in this paper are available in the supplementary resources or online at gjhunt.github.io/containerize/.

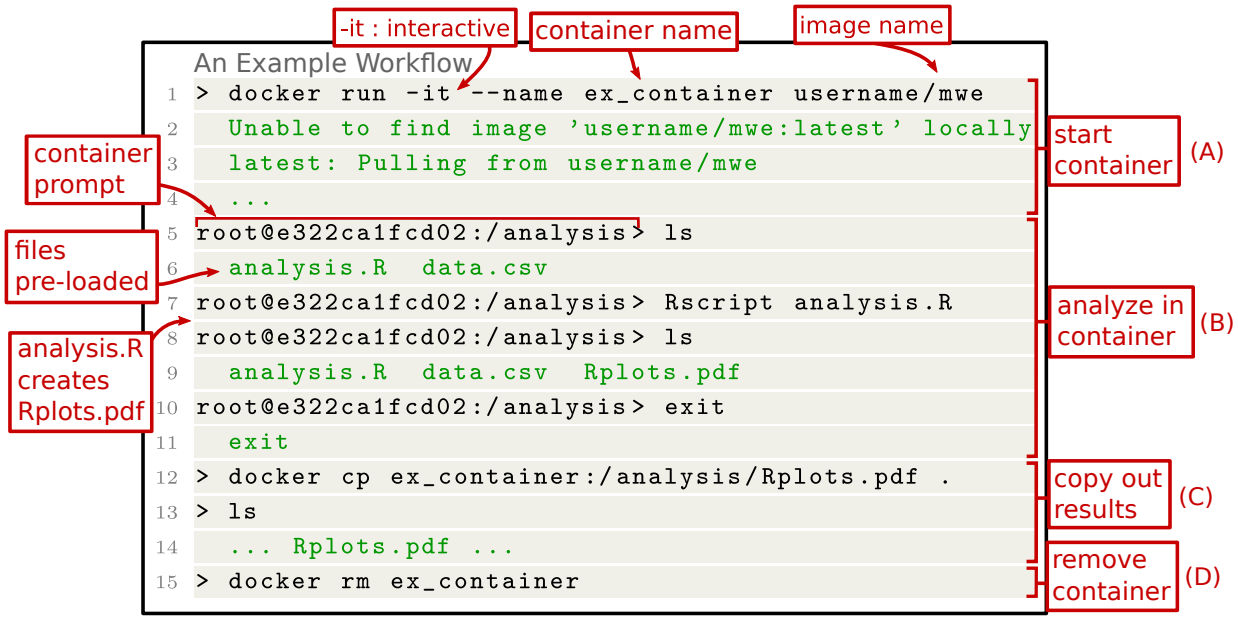

Figure 2: Example terminal input/output for running a container using Docker. Input is black and preceded by a caret `>`. Output is in green and truncated by a ellipsis. **A: (Lines 1-4)** `docker run` starts an example container `ex_container` from a hypothetical image `username/mwe`. For an actual image, "username" would be replaced with an actual user name. A runnable version of this example is available in the supplementary material. The flag `-it` opens an interactive connection to the container, otherwise the container will immediately pause. **B: (Lines 5-11)** We we are now in the container's environment as indicated by the change in prompt from "`>`" to "`.../analysis>`". Henceforth, all commands will be executed within the container's environment. `ls` shows that the container's environment contains a data file `data.csv` and an analysis script `analysis.R`. R is installed and can be used to run the script using the `Rscript` command to produce `Rplots.pdf`. The `exit` command leaves the container returning the user to the host environment. This merely pauses the container but does not remove it or its contents. **C: (Lines 12-14)** Thus `docker cp` can copy out the contents of the paused container. **D: (Line 15)** When done with analysis, `docker rm` removes the container, freeing up host resources.

that it does not needlessly consume the host computer's resources.

In addition to containers being easy to run, they are also easy to build. Setting up an image is analogous to setting up a new computer. However, containerization comes with several significant time-saving short-cuts. To create an image a configuration file must be written giving instructions of which files and programs to be copied and installed in the image. Importantly, this configuration file need not be built from the ground up. Instead, one can simply add on to existing pre-configured images to create new ones. This makes containerizing analyses quick and easy since one can choose a nearly-complete image, with desired software already installed, and simply add a small amount of project-specific code, data, and documentation. Repositories like dockerhub provide a wealth of images containing pre-installed software like R, python or julia (Docker Inc., 2021a). These images serve as great starting points upon which to build.

Figure 3 (A) displays an example configuration file. This is the exact configuration used to build the image from Figure 2. Far from being complex, the configuration file is only six lines long. The configuration specifies a base image with R already installed, installs a desired R package, and then copies over data and analysis code. Additionally, the image sets up a working directory for analysis. A working directory is good organizational practice as it specifies the directory in which the container starts and the default directory into which files are copied.

Such a simple configuration file is quite typical for containerizing statistical analyses. Most of the heavy lifting is done by the base image which sets up a nearly complete environment. On top of this base image one needs only to install the necessary software packages and copy over the data and code. Finally, once the configuration file has been written, the image needs to be built once, after which, it may be run or shared. Building the image is shown in Figure 3 (B). Intermediate steps are cached when building so if a change is made to the configuration only the commands from that point down in the file will need to be re-built.

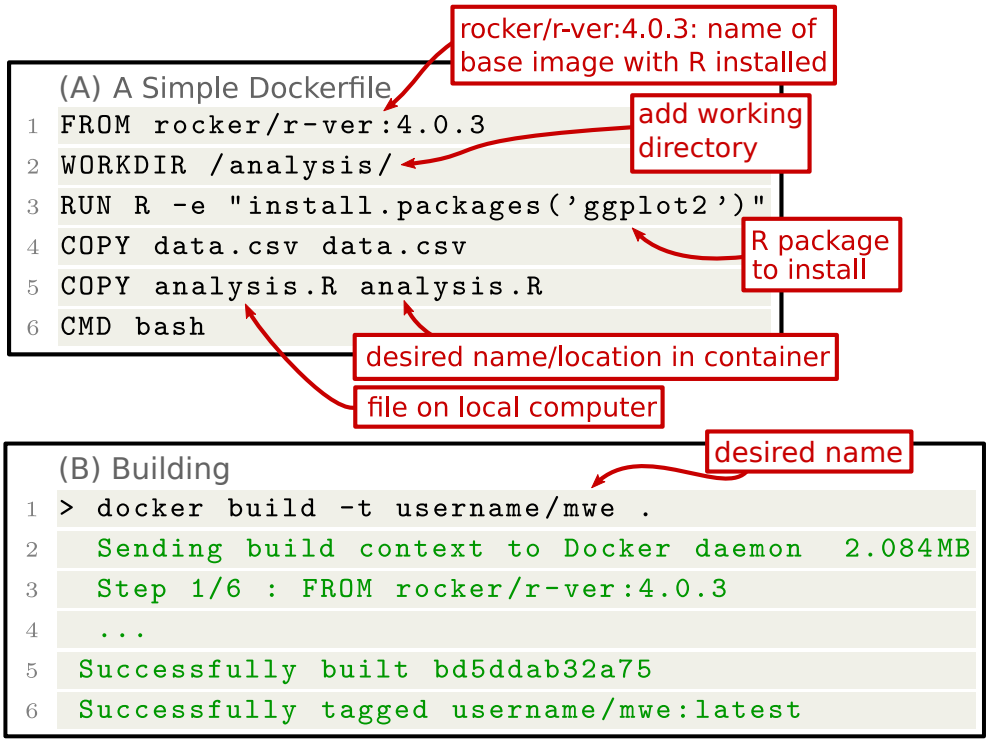

Figure 3: (A) Example configuration file for Docker. This file is traditionally named Dockerfile. **(Line 1)** `FROM` specifies the base image named `rocker/r-ver:4.0.3` to get a container with R version 4.0.3 already installed. **(Line 2)** `WORKDIR` sets up working directory as `/analysis`. **(Line 3)** `RUN` executes code which calls R and installs `ggplot2`. **(Line 4-5)** Copies the data and code. First argument to `COPY` is location on host, second argument is desired location in container (relative to `WORKDIR`). **(Line 6)** `CMD` sets the command executed when the container starts. Here, `bash` starts a Linux terminal. (B) Building image from Dockerfile. Flag `-t` specifies the image name as `username/mwe`. "." specifies necessary files to copy are in the current directory.

2.2 Sharing and Archiving

Once built, images may be saved locally as a file. For example, Docker saves images to a compressed file using `docker save`. Subsequently, these images may be shared just like any other file. For example, one can upload them to online storage platforms and share download links on institutional web-pages or in manuscripts. After downloading the image one can load it into their system using `docker load` and subsequently run the image. Alternatively, Docker images may be hosted on an online image repository. For example, Docker's native repository dockerhub allows images to be uploaded and downloaded to/from dockerhub using `docker push` and `docker pull`, respectively. Notice in Figure 2 that the image we run is automatically pulled from dockerhub because it does not exist locally on the machine. Dockerhub is highly analogous to github but for Docker images instead of raw code. However, public repositories like dockerhub and github are not ideal solutions for long-term archival of data-heavy images as hosting large amounts of scientific data is not their intended purpose. Indeed, github limits file sizes and dockerhub requires that images be accessed on a regular basis. However, there are still several options for archiving large images. We recommend that images are redundantly stored on both a local hard-drive and in an institutional cloud-based storage. Such cloud-based storage platforms have become ubiquitous at scientific institutions in recent years. In the future these institutions may even want to create their own image repositories akin to dockerhub, as this would further help facilitate the seamless sharing and archiving of reproducible analyses. This has already begun in some places, for example, users of the collaborative research management service synapse can host their data-intensive images on the private synapse docker repository (Sage Bionetworks, 2020). However, in the future, other institutions with a stake in the archival of analyses, such as journals, might consider hosting images as well.

3 Notebooks and Interactivity

Notebooks are a document format that allow interweaving of commentary, code, and output all together. Two primary benefits of this format are that it (1) promotes a natural organization of analysis and (2) helps create easy entry points into code. As an example,

Figure 4 displays a snapshot of conducting analysis using a code notebook in the JupyterLab integrated development environment (IDE). While there are several variants of code notebook formats, they are all structured as a sequence of "chunks" that can be edited and evaluated one at a time. Each chunk can either be text or code. Text chunks can typically include formatting using markdown notation, while evaluating code chunks runs the code and displays the text or plot output inline (Matt Cone, 2021). Some IDEs also allow embedding of interactive elements into the notebooks, although this is not supported by all software.

The chunked structure of notebooks naturally breaks up code and annotation into bite-size segments which helps organize analysis into a logical flow. The ability to interweave bite-sized chunks of code and text makes notebooks well suited for explaining, showcasing, and exploring data and analysis. Since chunks can be edited and run one at a time, each chunk provides a natural entry-point into a small portion of the analysis. For example, one can pick a segment of the analysis they wish to explore, edit the chunk of code, run it, and observe the subsequent change in output. This allows one to experiment with small changes to code, e.g., testing different tuning parameters or optional arguments to functions, and immediately observe the changes to local output without having to re-run the entire analysis. This provides a natural way to play with code in order to build up an understanding of how the code works and test the robustness of the analysis to alterations. For this reason, notebooks and their associated IDEs have become increasingly popular in the statistical community.

Several formats for code notebooks and associated IDEs exist. For writing notebooks in R, the RStudio IDE and its R Markdown format are popular (RStudio, PBC, 2021). For Python, a popular notebook format has been developed by Project Jupyter along with two IDEs: (1) an IDE called "The Jupyter Notebook" and (2) its successor IDE called "JupyterLab." (We will use "Jupyter" to refer collectively to the IDEs and reserve the term "Jupyter notebook" for the notebook format.) While Jupyter notebooks are popular for Python and R Markdown is popular for R, the notebook formats are actually language agnostic and one can write notebooks in either format using any of the most popular languages like R, Python, Julia, Octave, SAS or many others.

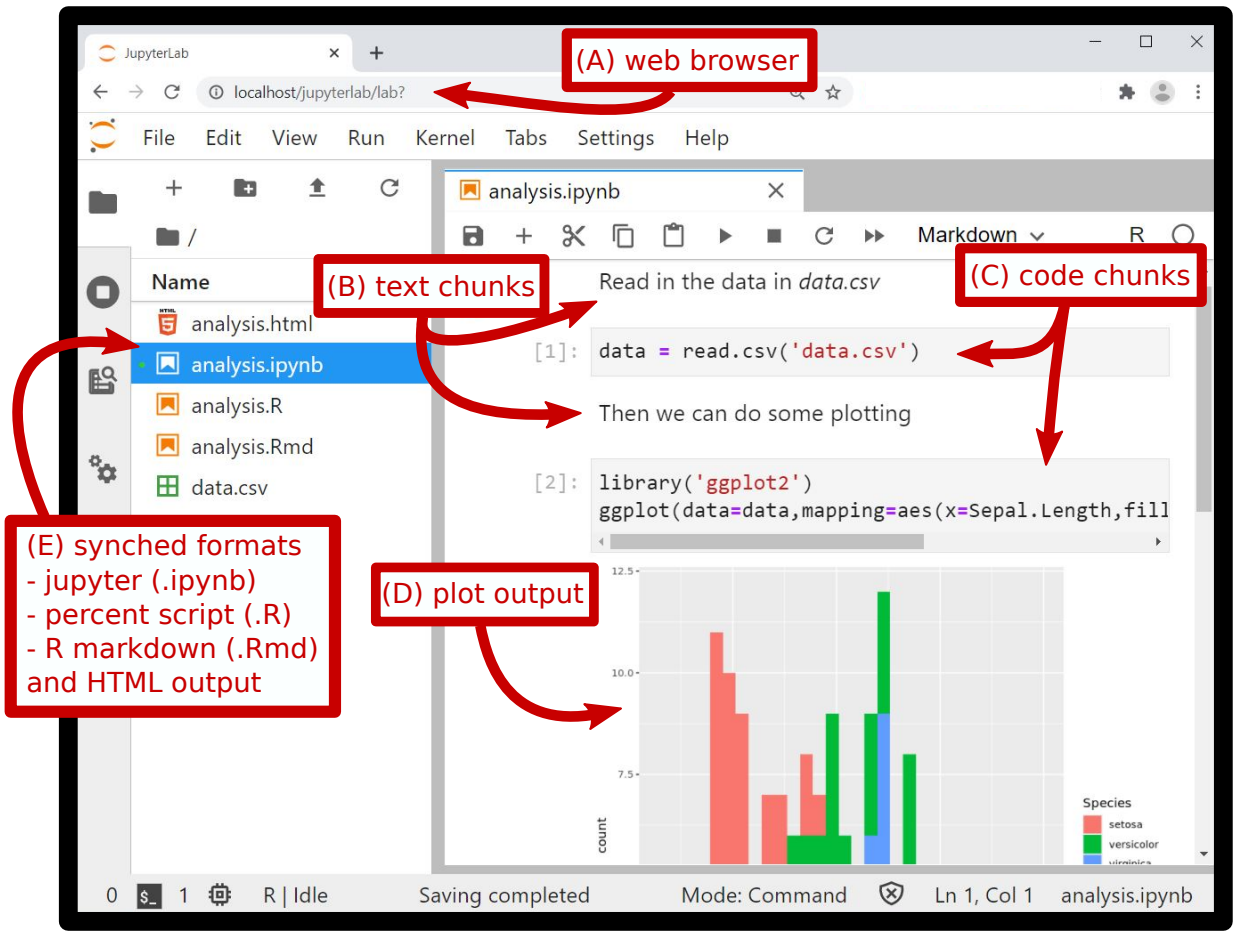

Figure 4: Analysis using a code notebook in JupyterLab. (A) JupyterLab can be accessed through the web browser. (B) Markdown can be added to text chunks which renders when evaluated. (C) Code chunks allow input, here in R, and (D) display associated output inline when evaluated. (E) The Jupyter notebook (analysis.ipynb) is synchronized to a mirrored R script (analysis.R) and R markdown file (analysis.Rmd) using the jupytext plugin. This ensures that changes made to the jupyter notebook will propagate to the mirrored script and markdown files. Notebook input and output can also be rendered in non-code formats like HTML or PDF for showcasing analysis. Here, we produce analysis.html for display on webpages.

Nonetheless, one major difference among notebook formats is whether they are saved as plain text or in a JSON format. The JSON format is used by Jupyter notebooks while R markdown uses plain text. Each format has advantages as well as drawbacks. JSON allows Jupyter notebooks to save output text and plots as well as input code and commentary. This is useful for showcasing results because one can simply open a Jupyter notebook in Jupyter and examine analysis output without having to re-run the code chunks. However, the JSON format saves much of the output using a dense encoding that is not readable by humans. This encoding can cause difficulties when combined with software other than Jupyter. For example, one cannot easily track changes to Jupyter notebooks using git and github as small changes to output can prompt a cascading change to hundreds of lines of densely encoded output. This significantly diminishes the ability of version control software to meaningfully document a human-readable record of changes to analysis. Alternatively, plain text notebook formats like R Markdown save input code and commentary in a human-readable format which is more versatile for general editing and can be meaningfully tracked by version control software. However, plain text formats cannot encode output and thus to display the analysis results one must either re-run all code chunks in an IDE or save the output separately (e.g., as HTML or PDF).

To get the best of both JSON-formatted and plain text notebooks, one can use the software jupytext, which can be used as a standalone program or as a plugin for Jupyter. jupytext allows one to conduct analysis using a JSON-format Jupyter notebook but maintain simultaneous synchronized versions saved in any of the popular plain text formats like R markdown. In Figure 4 (E) the Jupyter notebook is synchronized to mirrored plain text versions. Finally, in addition to saving the notebook itself, IDEs typically allow rendering of the notebook into popular non-code formats like PDF, markdown, or HTML. These renderings of the notebook create a snapshot of the input and output after running all of the chunks sequentially. While one cannot use these non-code formats to edit or run the code, they are ideal for showcasing the analysis and results, for example, as HTML for a webpage.

3.1 Containerizing Interactive Notebooks

Many popular notebook IDEs can be used interactively through a web browser. For example, JupyterLab natively runs in a web browser as shown in Figure 4. Similarly, R Studio also has a web browser version called R Studio Server. One advantage of such browser-based IDEs is that they can be used to provide a graphical user interface for containers. An example of this is displayed in Figure 5. Subplot (A) displays the configuration file for our image. Here, we build off a base image we have created containing Python, R, Jupyter, R Studio Server, and R Shiny. (R Shiny is a a platform for building interactive web apps in R for showcasing results.) While our base image contains several popular statistical tools, other base images like `jupyter/datascience-notebook` provide similar interactive functionality and could be used instead. On top of this base image we have added some data and a Jupyter notebook analyzing the data. We have also used jupytext to sync the Jupyter notebook to an R markdown and R script version. Additionally, we have rendered an HTML version of the notebook for showcasing.

After building the image, Figure 5 (B) demonstrates how the container may be started. Once running, the container is accessible through the host computer's web browser (subplot (C)). The initial start page for the container offers several options. One may browse the container's files (subplot (D)) and, for example, view the HTML rendering that showcases the analysis. Alternatively, one may choose among several browser-based IDEs to conduct analysis. The start page offers links to The Jupyter Notebook, JupyterLab or RStudio which open the corresponding IDEs in new tabs (subplot (E)). These IDEs can be used to interact with the data and notebooks added to the container. Thanks to jupytext, one may open the analysis in any of several synchronized choices of notebook format: Jupyter notebook, R Markdown, or R script.

While we access this container frontend through the web browser on the host computer, the data, code, and IDE backends all actually reside in the container. The web browser merely provides a window into the running container through which one may use the IDEs installed in the container to interact with the notebooks and data it contains. Indeed, none of these IDEs need be installed on the host computer in order to use the web browser to interactively access the versions running in the container. This is the power of containerizing

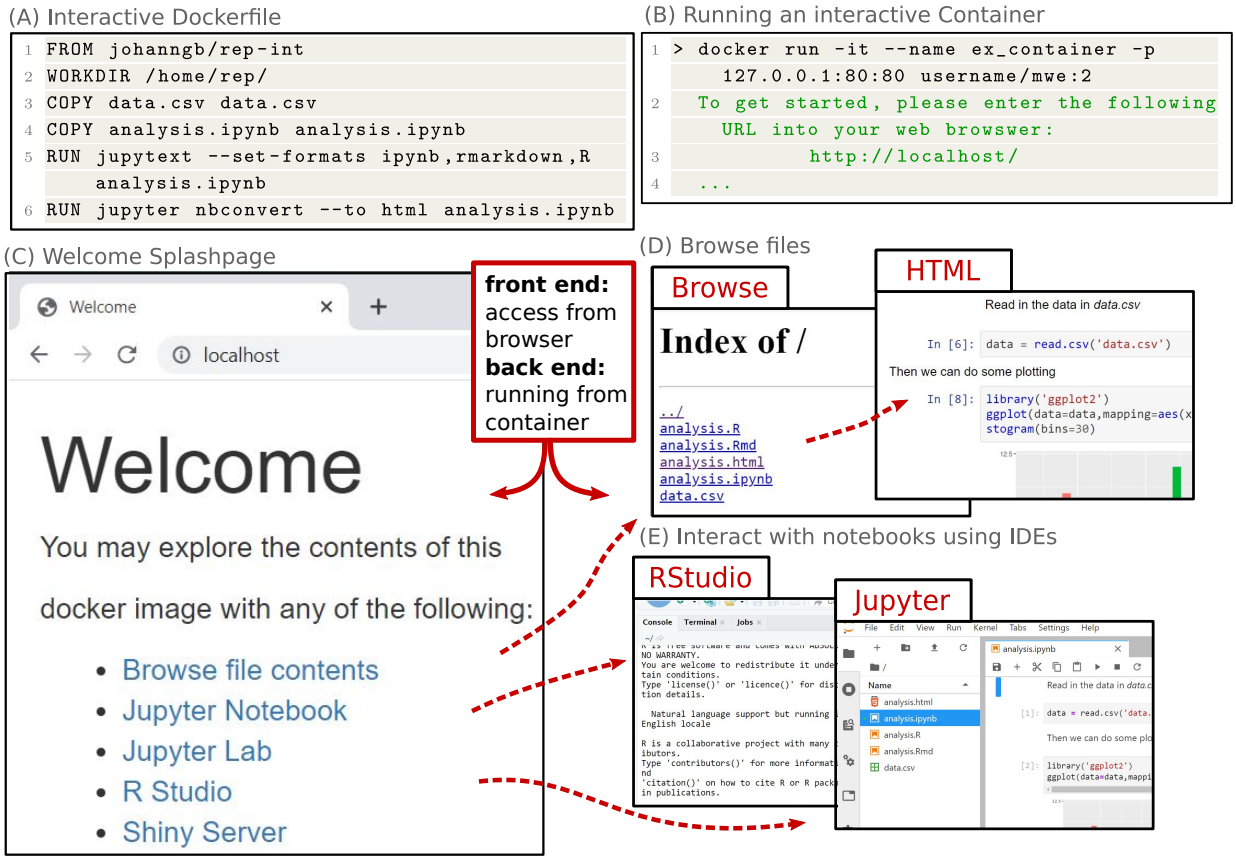

Figure 5: Building an interactive container accessible through a browser on the host computer but with a backend running in the container. (A) **(Line 1)** An interactive dockerfile built from `johanngb/rep-int` base. **(Line 5)** `jupytext` links the .ipynb to the .Rmd and .R versions of the notebook. **(Line 6)** `jupyter` runs the notebook and saves input/output as a HTML document for showcasing. (B) We build the image and name adding the tag `:2` to indicate it is version 2 of our previous example. We then start up the container using `-p` to correctly map ports between the container and host. Note that by specifying 127.0.01 we ensure that the computer doesn't expose itself to the wider network but still allows us to access the container from a web browser on the host by putting in `localhost` for the URL. (C) The start page for the interactive container. Several options for interacting with the analysis files are listed. (D) We may browse the files or (E) open the notebooks with one of several choices of graphical web-based IDEs running from the container.

code notebooks along with a graphical web-based IDE. It allows users to bring to bear the full power and convenience of several popular graphical IDEs to containerized notebook analyses. This containerization ensures that not only are the data, notebooks and code dependencies reproducibly configured, but the IDEs, too, are containerized and thus work out of the box.

3.2 Chunking and Caching

In addition to enabling a reproducible and interactive environment, containerizing of code notebooks allows creation of practical entrypoints into the analysis. In Figure 6 (A) we present an example notebook that builds and tests a simple k-NN classification model. The first chunk represents some time-consuming data pre-processing, the second chunk builds the model, and the third chunk evaluates model accuracy. In this notebook, the chunked structure logically separates the time-consuming first steps from the quicker model building and evaluation. This makes it easy to play with the code. For example, one may wish to evaluate the robustness of the model to changes in the number of neighbors `k`. To do this, one may change `k`, re-run the second and third chunks, and observe changes to the accuracy. The notebook structure promotes quick and iterative testing of different values of `k` accompanied by immediate in-line feedback to those changes.

While such chunked structure helps promote logical entry points into code, time-intensive code can still pose practical challenges for third-parties. For example, one may wish to open the notebook in Figure 6 (A) and immediately try various the values of `k`. However, the initial time-consuming chunk must first be run. For real analyses, it is not uncommon for there to be processing chunks that take hours or days to compute. Thus, while technically reproducible, such analyses are impractical to re-run. To overcome this challenge, we advocate for containerizing pre-computed intermediate results alongside the notebooks. Instead of re-running computationally expensive chunks, one can instead load the pre-computed results and proceed with analysis. This practice is known as results caching. In Figure 6 (B) we present an example of caching intermediate results. The caching code simply checks to see if pre-computed results already exist and loads them if so. Otherwise, it re-runs the long calculation and then caches (saves) the results in a file. This caching idiom could

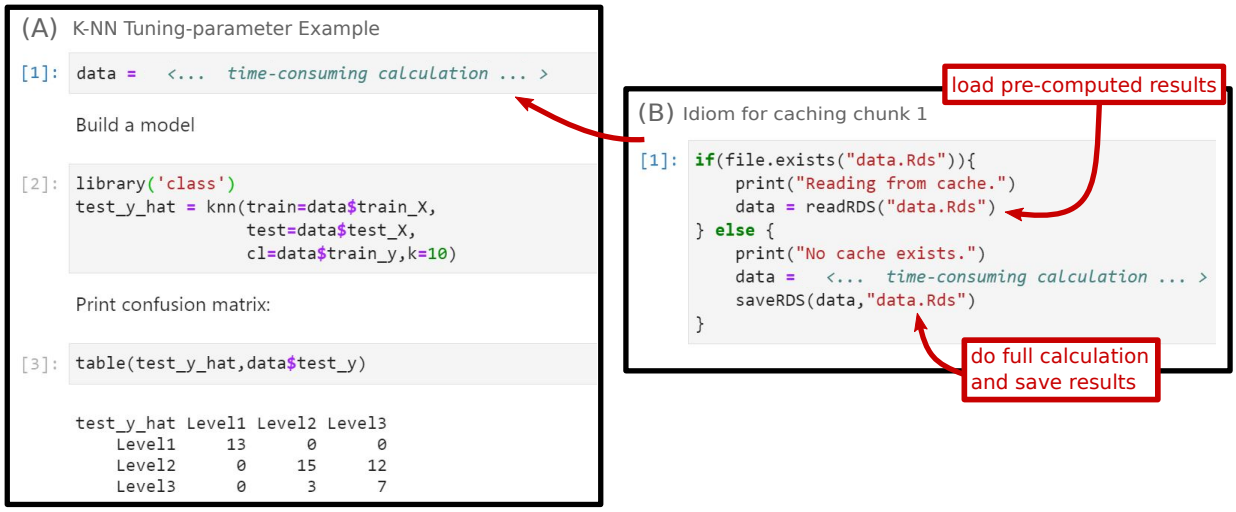

Figure 6: (A) An example notebook for building and evaluating a simple k-NN model. The first chunk represents a time-consuming data pre-processing step, the second builds the model and the third evaluates. (B) A programming idiom for caching intermediate results. This implementation in R makes use of the `readRDS` and `saveRDS` to serialize and save the data object. Similarly idioms can be implemented in other languages e.g. in Python using the `pickle` library.

be used in place of the first chunk in subplot (A). This would allow one to simply load the pre-computed results of the first chunk and immediately proceed to assessment of the model in subsequent chunks. If one wishes to force re-computation of the first chunk, they can simply delete the cache file and re-run that chunk.

Containerized notebooks provide a format highly suitable for this type of caching since they allow packaging of notebooks together with files containing intermediate results. More broadly, one can easily create multiple tiers at which third parties may enter into analyses all the way from raw sources, to cleaned and processed datasets, to final results, plots, and documentation. One may interact with any of these tiers within the container's environment or copy out intermediate results and code to remix into new projects and analysis.

4 Code Sharing and Beyond

Writing and sharing code has become an increasingly important component of modern statistical analyses. We advocate for sharing containerized code notebooks as it can help ensure that code can (1) be replicably run across different machines, and (2) be easily understood, altered, and experimented with. Such an approach has potential benefits in a wide variety of contexts. For example, containerizing interactive code notebooks has the potential to enhance the efficacy and efficiency of the peer review process. Here, in addition to asking that authors provide code scripts and data for their analysis, journals might ask authors to provide an interactive containerized version of their analysis notebooks. Building such an image is relatively easy for the authors to do and would allow reviewers to quickly and painlessly jump into a working version of the analysis. Such an approach would make it easy for reviewers to quickly modify and re-run the analyses. This makes it more straight-forward to evaluate the robustness of a proposed method, allowing reviewers to easily play with the code, changing tuning parameters, function arguments, or software versions and observing the resulting differences in output. Containerization can help make such a comprehensive peer review more accessible.

In addition to efficiency and efficacy, containerized analyses provide a more secure way for reviewers to run third party code. Without explicit permission, containerized code cannot see, change, affect, or in any way alter other containers or the host system. Thus

reviewers can be assured that the effects of assessing containerized code will be limited to the container. For security-conscious institutions and organizations this may be attractive because it helps users avoid running unverified third party code directly on their systems. Beyond security concerns, containerization also means that one need not clutter up their system in order to evaluate third party analyses. Since containers come with all dependencies neatly packaged together, a reviewer need not install obscure packages on their system that they will never use again.

Containerized code notebooks may also be used as a tool for teaching. Containers allow distribution of code, data, and a computing environment in a way that ensures all students have identical and correctly configured workspaces. This can help avoid troubleshooting complications and save time. Conversely, student projects in applied statistical courses could be containerized before submission. While a small amount of time would need to be devoted to teaching students some simple mechanics of containerization, in our estimation this is not more complicated than other coding tasks required in many courses. Furthermore, this would provide an opportunity for a discussion with students about research reproducibility, replicability as well as good practices to adopt for coding and analysis. Certainly such discussions are warranted in advanced undergraduate and graduate-level courses.

Beyond the direct benefits of making code more easily shareable, our experience is that the act of containerizing analyses can itself serve as a helpful review step in a scientific pipeline. Preparing analyses for containerization forces one to review the code. This encourages simplification and refactoring of code, as well as writing of the associated documentation and commentary. Furthermore, containerized code will ideally be able to reproduce analysis from raw data all the way through final output, re-creating necessary intermediate steps along the way. Containerizing such a pipeline forces all analysis decisions to be documented by the code and ensures that all intermediate code and data are included in the image. If one forgets to include a data file, script, or analysis step in the image, the containerized analysis will not run. This serves as a strong check that a completely reproducible pipeline has been created.

Additionally, software like Docker can be interwoven seamlessly into popular code shar-

ing and versioning workflows. For example, one can connect github and dockerhub accounts together so that updates to code on github are automatically propagated to dockerhub where an image is subsequently built. Alternatively, Docker can directly pull and build repositories from github. Thus, containerization of workflows can be seamlessly integrated with collaborative versioning tools that help maintain a detailed record of the evolution of analysis.

In closing, we would again like to emphasize that containerization is more than just an approach for preserving passive code archives. It allows rich interaction and exploration of analysis and helps create usable and reproducible analyses. We believe that such an approach can enhance the ability of statisticians to easily share code, analyses, and ultimately ideas.

References

Docker Inc. (2021a). Build and Ship any Application Anywhere. https://hub.docker.com/.

Docker Inc. (2021b). Empowering App Development for Developers — Docker. https://www.docker.com/.

Ellis, S. E. and J. T. Leek (2018). How to share data for collaboration. *The American Statistician 72*(1), 53–57. PMID: 32981941.

Felter, W., A. Ferreira, R. Rajamony, and J. Rubio (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172.

Github, Inc. (2021). Where the world builds software. https://github.com/.

Kaggle Inc. (2019). Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/.

Matt Cone (2021). Markdown Guide. https://www.markdownguide.org/.

National Academy of Sciences, National Academy of Engineering, and Institue of Medicine (2009, apr). *On being a scientist: A guide to responsible conduct in research: Third edition*. National Academies Press.

Project Jupyter (2021). Project Jupyter — Home. https://jupyter.org/.

Python Software Foundation (2021). PyPI · The Python Package Index. https://pypi.org/.

RStudio, PBC (2021). RStudio — Open source & professional software for data science teams - RStudio. https://rstudio.com/.

Sage Bionetworks (2020). Synapse Docker Registry. https://docs.synapse.org/articles/docker.html.

The R Project for Statistical Computing (n.d.). The Comprehensive R Archive Network. https://cran.r-project.org/.