Dependency hell

Tad Dallas

Reading for this week:

> Boettiger, C. (2015). An introduction to Docker for reproducible research. ACM SIGOPS Operating Systems Review, 49(1), 71-79.

What are dependencies?

We have covered dependencies a bit throughout this course, with the general approach that we should limit the number of `R` packages that our analytical workflow relies on. *Dependencies* refers to these external packages which we rely on. There are a number of issues with including too many dependencies.

Dependencies can change

Your code from 10 years ago likely depends on packages that have almost certainly been updated and changed. This may break your code, if the package maintainers were not nice enough to make their new code backward compatible (hint: the maintainers are mostly us, and we have no idea what we are doing).

Dependencies have dependencies

`R` packages have dependencies (this is normally how the term 'dependency' is used in the Rstats community in my experience). These are other `R` packages which provide essential functionality to the package. This becomes pretty apparent when we try to issue a command like

```
install.packages('tidyverse', dependencies=TRUE)
```

```
## Installing package into '/home/tad/R/x86_64-pc-linux-gnu-library/3.6'
## (as 'lib' is unspecified)
```

```
## also installing the dependency 'feather'
```

This installs not just the 8 core tidyverse packages, but also any packages that these packages depend on. How does this look?

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.1     v purrr   0.3.4
## v tibble  3.0.2     v stringr 1.4.0
## v tidyr   1.1.0     v forcats 0.5.0
## v readr   1.3.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::arrange()   masks plyr::arrange()
## x purrr::compact()   masks plyr::compact()
```

```
## x dplyr::count()     masks plyr::count()
## x dplyr::failwith()  masks plyr::failwith()
## x dplyr::filter()    masks stats::filter()
## x purrr::flatten()   masks jsonlite::flatten()
## x dplyr::id()        masks plyr::id()
## x dplyr::lag()       masks stats::lag()
## x dplyr::mutate()    masks plyr::mutate()
## x dplyr::rename()    masks plyr::rename()
## x dplyr::summarise() masks plyr::summarise()
## x dplyr::summarize() masks plyr::summarize()
```

```
sessionInfo()
```

```
## R version 3.6.3 (2020-02-29)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.4 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.2.20.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] forcats_0.5.0   stringr_1.4.0   purrr_0.3.4     readr_1.3.1
##  [5] tidyr_1.1.0     tibble_3.0.2    ggplot2_3.3.1   tidyverse_1.3.0
##  [9] rgbif_3.1.0     jsonlite_1.7.0  httr_1.4.1      dplyr_1.0.0
## [13] plyr_1.8.6
##
## loaded via a namespace (and not attached):
##  [1] maps_3.3.0        bit64_0.9-7       modelr_0.1.8      assertthat_0.2.1
##  [5] sp_1.4-2          triebeard_0.3.0   urltools_1.7.3    highr_0.8
##  [9] blob_1.2.1        cellranger_1.1.0  yaml_2.2.1        pillar_1.4.4
## [13] RSQLite_2.2.0     backports_1.1.7   lattice_0.20-41   glue_1.4.1
## [17] uuid_0.1-4        digest_0.6.25     rvest_0.3.5       colorspace_1.4-1
## [21] htmltools_0.4.0   pkgconfig_2.0.3   oai_0.3.0         httpcode_0.3.0
## [25] broom_0.5.6       haven_2.3.1       xtable_1.8-4      scales_1.1.1
## [29] whisker_0.4       generics_0.0.2    ellipsis_0.3.1    withr_2.2.0
## [33] lazyeval_0.2.2    cli_2.0.2         magrittr_1.5      crayon_1.3.4
## [37] readxl_1.3.1      memoise_1.1.0     evaluate_0.14     fs_1.4.1
## [41] fansi_0.4.1       nlme_3.1-147      xml2_1.3.2        tools_3.6.3
## [45] data.table_1.12.8 hms_0.5.3         lifecycle_0.2.0   reprex_0.3.0
## [49] munsell_0.5.0     geoaxe_0.1.0      compiler_3.6.3    tinytex_0.23
## [53] rlang_0.4.6       grid_3.6.3        conditionz_0.1.0  rstudioapi_0.11
## [57] tcltk_3.6.3       rmarkdown_2.2     testthat_2.3.2    gtable_0.3.0
## [61] DBI_1.1.0         curl_4.3          R6_2.4.1          lubridate_1.7.9
```

```
## [65] knitr_1.29        rgeos_0.5-3       bit_1.1-15.2      utf8_1.1.4
## [69] stringi_1.4.6     crul_0.9.0        Rcpp_1.0.5        vctrs_0.3.1
## [73] dbplyr_1.4.4      tidyselect_1.1.0  xfun_0.15
```

yikes.

Dependencies may be OS sensitive

We would like to think that since `R` can be installed across operating systems (OS), that packages built on top of `R` would also have that flexibility. This is not true. For example, the `osrm` package requires that you build a local instance of the OpenStreetMap-Based Routing Service (OSRM), which is dependent on OS. Another example is in `rstan`, which – like many other R packages with a C++ backend – require different instructions for installation for different operating systems. And the way dependencies work, this bug will not only affect the `osrm` and `rstan` packages, but any packages which rely on the successful installation of these packages. Now we can see how the web of dependencies among `R` packages can cause some issues.

What is dependency hell?

Dependency hell is not just about this inter-related network of package dependencies that could hamper the reproducibility of analytical pipelines.

It can take on a few different forms that – if and when you encounter them – will infuriate you.

1) Packages or some software relies on specific versions of other software. You may have hit this with `R` packages depending on a newer version of `R` than you have installed. This is a relatively easy fix. When this becomes irksome is when two different softwares rely on the same underlying software, but different versions (just ran into this with tensorflow, python, and some other software). So if you upgrade or downgrade to make one software work, the other software may break.
2) Many dependencies. Some packages have a super long list of dependencies (e.g., `devtools`) and takes a bit of time and memory to install all that mess.
3) Dependency chains. The packages which a package depends on may have further dependencies. This creates a situation where the user must download and configure numerous packages thanks to the dependencies of a dependency.

Claes et al. 2014 found that "occurrence of errors in CRAN packages over a period of three months ... resulted mostly from updates in the packages' dependencies ... "

> Claes, M., Mens, T., and Grosjean, P. (2014). On the maintainability of CRAN packages. 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). https://doi.org/10.1109/csmr-wcre.2014.6747183

How do I avoid dependency hell?

Use fewer dependencies

A clear way to stay out of dependency hell is to simply use fewer dependencies. This may simply not be feasible for some projects (e.g., geospatial data racks up dependencies pretty quickly). If you are only using one function from a package, consider reverse engineering it or checking out their source to see if you can simply include it in your workflow without requiring it to be called in with the `library` function. There are some good articles on code dependency and this semi-minimalist mentality (http://www.tinyverse.org/), and also some more general discussion about the nuances of dependency management (https://rstudio.com/resources/rstudioconf-2019/it-depends-a-dialog-about-dependencies/).

Be explicit about packages and versions

Packages are versioned quite effectively using semantic versioning. This is an approach which provides information package history and version, and comes in the form of "majorVersion.minorVersion.patch". At the time of writing these notes, the `R` package devtools was at version 2.3.0. Semantic versioning is good practice, and essentially tags package versions that can then be targetted for specific installation. How does this help with reproducibility? We can specify the `R` package versions that we would like to install, essentially making sure that no changes to the packages could break functionality. Where this fails is when the installation of a previous version of a package fails to install the correct version of the dependencies, breaking core functionality despite being the correct package version.

```
cranUrl <- "http://cran.r-project.org/src/contrib/Archive/vegan/vegan_2.0-0.tar.gz"
install.packages(cranUrl, repos=NULL, type="source")
```

This installs the `vegan` package version 2.0.0 from the year 2011 to my computer. The obvious downside to this is that now I have a super outdated version of `vegan`, and other packages installed on my machine that rely on `vegan` (e.g., `metacom`) are broken. There is a nice way around this, which I discuss below, and have been told by the internet and interactions with colleagues is not worth trying to teach at this level given the breadth of material already covered. The way around this is called "containerization". Basically, if you do not want to install outdated packages on your machine that will break existing functionality, one solution is create a separate "container" with some basic OS that is separate from your existing OS. More on that below.

side note: adding the function `sessionInfo()` to the end of your `Rmd` file will output relevant information on your OS, R version, and different packages that are loaded into the workspace. While this places the onus on the reader to try to mirror your settings, it is a nice addition to a Rmd file.

Use a service

Before we talk about containerization a bit further, I would like to note that there are existing packages to handle dependencies. There are two main packages for package management: `packrat` and `renv`. I will only go over `packrat`, as they both use similar concepts, but there are some neat things about `renv` that may make it "better".

```
packrat::init("myProject")
```

After this command is issued, any packages installed from inside this packrat project are **only available to that project**. Updates to the existing `packrat` state are performed using the `packrat::snapshot` function.

Starting `R` from the directory (`myProject`) will automatically load the packages in their specific versions required for your analytical workflow. So provided that the end user has `R` installed and has `packrat` installed, the rest of the analytical pipeline should be *fairly* reproducible (barring any OS-specific issues, etc.).

Build the OS from scratch

This can be in at least two ways. First, we could use a virtual machine to set up an operating system on our local machine. Imagine this like opening up an internet browser, but instead of a browser, the window contains an entirely separate operating system, with file structures and software independent of the host OS. This can be incredibly useful. If your analytical pipeline also contains a VM image, this could be ensure reproducibility. How this would work is you would set up a virtual machine image of whatever OS you have access to (e.g., linux OS is nice because it is free and well supported), set up the image as persistent (meaning that files created within the image and software installed in the image are available after the instance of the OS is shut down), and then making sure all the dependencies specific to the project are installed and versioned. Boom.

Why is this not the norm?: because it is a big pain! It also assumes that the user has some form of virtual machine software like VMware or VirtualBox, that the user is comfortable in whatever OS the virtual

machine is in, etc. As we can sort of imagine, the layers of complexity to ensure reproducibility may also reduce accessibility to the majority of people who might want to run your code. This is a huge pain point, and something that has prevented me from taking some of these steps (e.g., using Docker images) in my own research.

Docker images are a slightly more simple approach to virtualization, in that the containerization approach of Docker reduces the huge computational overhead in using Virtual Machines, as having two operating systems running with separate software and shared hardware is pretty demanding. Containers bypass this issue by sharing the OS kernel of the host machine. More information on this idea and the differences between containers and virtual machines is available at https://www.docker.com/resources/what-container.

Docker

First things, first. Install `Docker` on your machine.

Docker has some associated language around it which can be confusing. A DockerFile is a text file which contains the instructions on how to build the subsequent DockerImage. These images can be stored on places like Docker Hub (https://hub.docker.com/). This is like Github but only for Docker Images. These DockerImages can be run as DockerContainers, which are the realizations of all the setup, allowing you access to different operating environments on your local machine, and ensuring that your code can be run on any machine that has Docker installed.

```
sudo docker pull r-base:latest
```

```
sudo docker images
```

The output of this on my machine looks like this:

```
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
r-base                      latest              53c050954e1f        3 weeks ago         783MB
julia                       latest              183ba11b1cb8        3 years ago         368MB
selenium/standalone-firefox 2.53.0              0067b27644ee        3 years ago         599MB
hello-world                 latest              690ed74de00f        4 years ago         960B
```

Each of the images can be run to form a container. If the container is interactive, we have an environment where commands can be issued that allow access to programs and such potentially independent of our computer operating system and structure. For instance, a computer without `R` installed could still run `R`.

```
sudo docker run -ti --rm r-base
```

Outputs

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
```

```
Type 'q()' to quit R.

>
```

This puts us into an `R` environment of a more recent version of `R` than what I have installed locally (which is my bad for not upgrading my `R` instance). None of the `R` packages I have installed on my local machine are accessible by this container, such that I have to install all the packages and versions to use in a given analysis. This also allows us to use older versions of software. For instance, based on the table above, I have a Docker image of julia (another statistical programming language) from 3 years ago, corresponding to version 0.5.0. The current version of julia at the time of writing these notes was version 1.4.2.

We can list running containers using the `docker ps` command (which should return no containers since we shut down the container above, treating it as interactive).

```
sudo docker ps
```

```
# stop the container
docker container stop

# remove container
docker container rm

#remove entire image
docker container rmi r-base

# remove all stopped containers
docker container prune
```