Manage Dependencies with Drake

Ben Herndon-Miller

1/31/2019

Basic Example

This introductory example was taken from the Drake GitHub repository: https://github.com/ropensci/drake. We are going to build a histogram and run a linear regression on the classic *iris* dataset that you have suredly encountered before. By embedding our data pipeline structure in the drake_plan() function we are able to manage our workflow, avoid unnecessary computation, and ensure our results are reproducible.

Load packages

We load the necessary packages to perform this module. Install any of these packages that you do not have using install.packages().

```
library(drake)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)
```

Load custom functions

Before creating our drake_plan(), we load our user-defined functions. These could be data cleaning or loading scripts, as well as any other functions you need to write for your project.

```
create_plot <- function(data) {
  ggplot(data, aes(x = Petal.Width, fill = Species)) +
    geom_histogram()
}
```

Check any supporting files

It is always good to make sure that the files you need are where you think they are before trying to run your pipeline.

```
# Get the files with drake_example("main").
file.exists("main/raw_data.xlsx")
```

```
## [1] TRUE
```

```
file.exists("main/report.Rmd")
```

```
## [1] TRUE
```

Plan what you are going to do

This is where we build our data pipeline. This function saves all of our data pipeline steps as objects called *targets* that maps to a command or operation that is to be performed. **Your drake plan is like a top-level R script that runs everything from end-to-end**.

It is important to note that targets are built in the correct order regardless of the row order in the drake_plan() function.

```
plan <- drake_plan(
  raw_data = readxl::read_excel(file_in("main/raw_data.xlsx")),
  data = raw_data %>%
    mutate(Species = forcats::fct_inorder(Species)),
  hist = create_plot(data),
  fit = lm(Sepal.Width ~ Petal.Width + Species, data),
  report = rmarkdown::render(
    knitr_in("main/report.Rmd"),
    output_file = file_out("main/report.html"),
    quiet = TRUE
  )
)
```

```
## Warning: Converting double-quotes to single-quotes because the
## `strings_in_dots` argument is missing. Use the file_in(), file_out(), and
## knitr_in() functions to work with files in your commands. To remove this
## warning, either call `drake_plan()` with `strings_in_dots = "literals"` or
## use `pkgconfig::set_config("drake::strings_in_dots" = "literals")`.
```

```
plan
```

```
## # A tibble: 5 x 2
##   target   command
##   <chr>    <chr>
## 1 raw_data readxl::read_excel(file_in('main/raw_data.xlsx'))
## 2 data     raw_data %>% mutate(Species = forcats::fct_inorder(Species))
## 3 hist     create_plot(data)
## 4 fit      lm(Sepal.Width ~ Petal.Width + Species, data)
## 5 report   "rmarkdown::render(knitr_in('main/report.Rmd'), output_file = ~
```

Execute data pipeline

We use the make() function in order to execute the plan we created in the previous step. This function runs all of the pipeline steps we defined in the drake_plan() function in the correct order. If we make any changes to the pipeline, they will be stored and updated automatically next time we call the make() function. The make() function does not re-run models that have not changed which saves us computational resources; this is especially helpful if we are performing lots of expensive computations.

```
make(plan)
```

```
## target hist
```

```
## target report
```

Read stored data

We can access any of the targets stored in the drake_plan() meta object by using the readd() function. Below we can see the dataset we will be performing our analysis on.

```
readd(data)
```

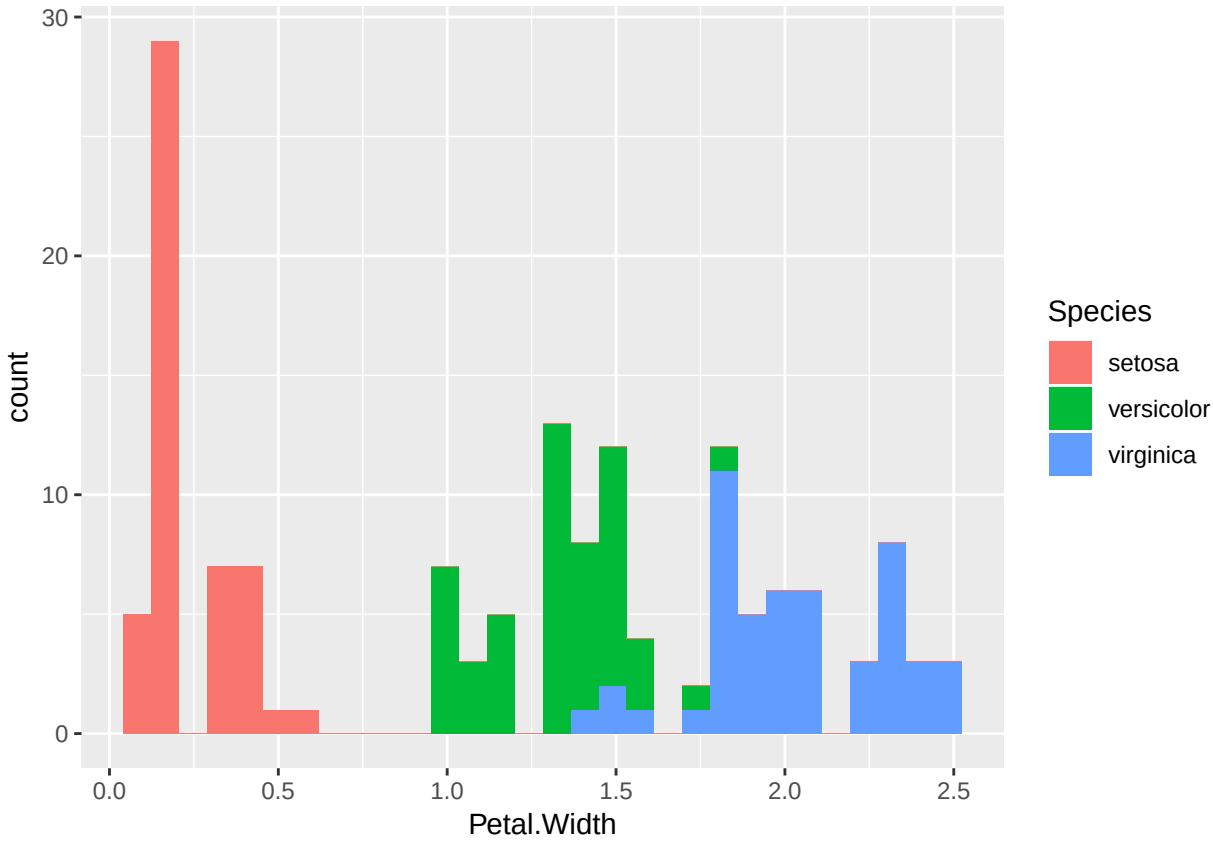
```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Read stored histogram

Just like when we accessed the stored target for the dataset, we can access target objects of other types as well, like a histogram in this case. Again, we just use the readd() function to access the target.

```
readd(hist)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Update plotting function

What happens when we update an object that is upstream in our data pipeline? How will it affect other parts of our pipeline? How will we keep track of what code we need to re-run?

Don't worry about it! That's what the drake library offers us. It automatically keeps track of changes in your pipeline and dictates what parts of your pipeline need to be re-run!

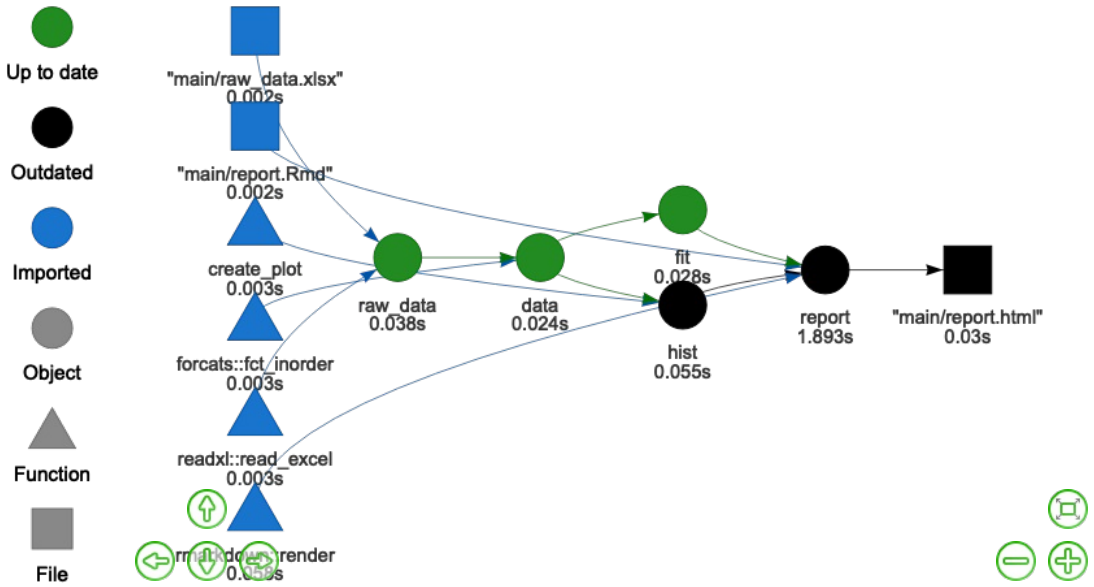
```
create_plot <- function(data) {
  ggplot(data, aes(x = Petal.Width, fill = Species)) +
    geom_histogram(binwidth = 0.25) +
    theme_gray(20)
}
```

Check what results have been affected

We can visualize our dependency structure using the drake_config() and vis_drake_graph() functions. Drake automatically generates this visualization that details what different components of the pipeline are and what is up-to-date and what is out-dated.

```
config <- drake_config(plan)
vis_drake_graph(config) # Interactive graph: hover, zoom, drag, etc.
```

Dependency graph



Re-run only outdated parts of pipeline

After making changes, we can just re-run the make() function to update our pipeline. Drake automatically keeps track of the changes that are made and only re-runs the parts of the pipeline that are out-dated. You can tell by the ouput below.

```
make(plan)
```

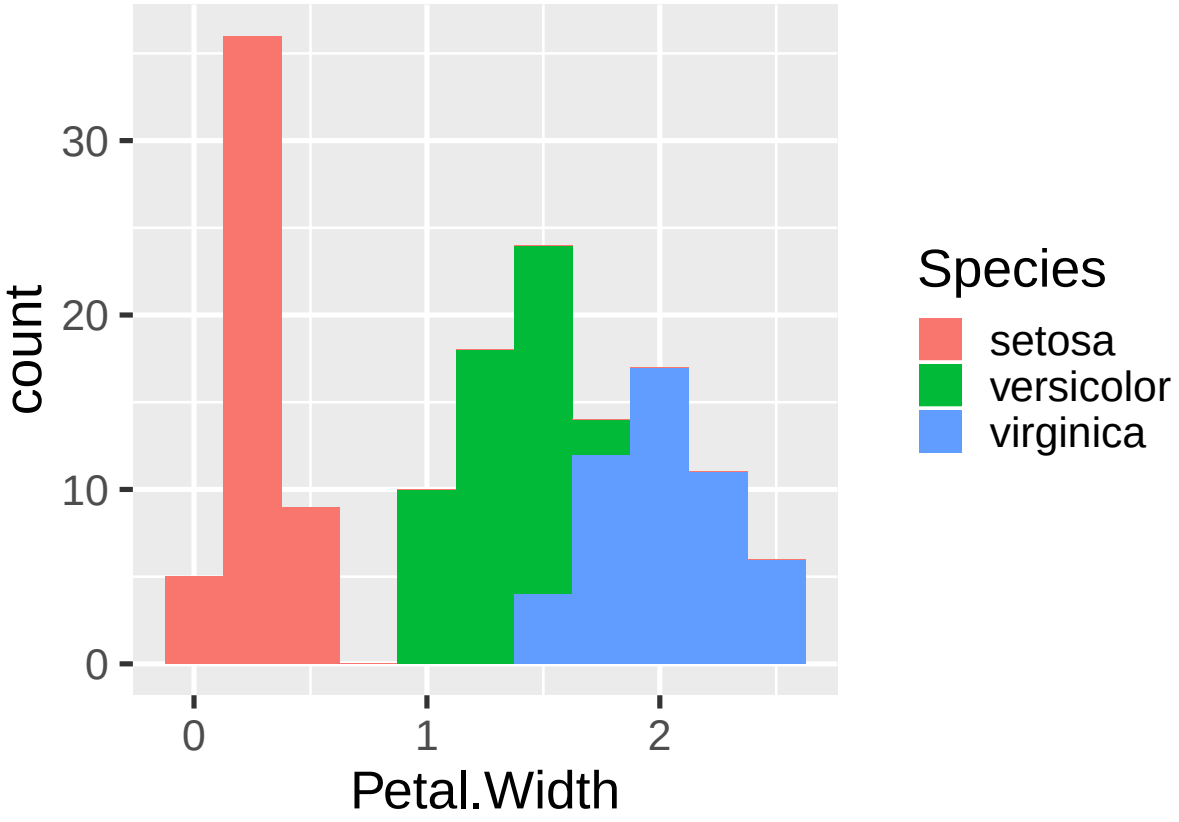
```
## target hist
```

```
## target report
```

Check that changes have been made on histogram

You can see the changes we made before reflected in the new histogram rendered below.

```
loadd(hist)
hist
```



Data Wrangling Example

Now that we have illustrated how Drake works on a basic example, let's see how we can apply it to the work that we did in the Data Cleaning and Wrangling module.

Call R scripts with our user-defined functions

Recall the work we did previously in the Data Cleaning and Wrangling module. We modularized functions to load our data and then sourced the necessary R script to source those functions. We shall do the same thing here.

```
source("cleaning_and_wrangling.R")
```

Make Drake plan

Now that we have called our user-defined functions, we can make our drake plan to map out our dependency structure.

```
dcw_plan <- drake_plan(# Load cleaned drinks dataset
  drinks_clean = clean_drinks(),
  life_exp_clean = clean_life_exp(),
  model_data_factor_clean = load_factor_data(),
  model_data_dummy_clean = load_dummy_data()
  )
```

```
make(dcw_plan)
```

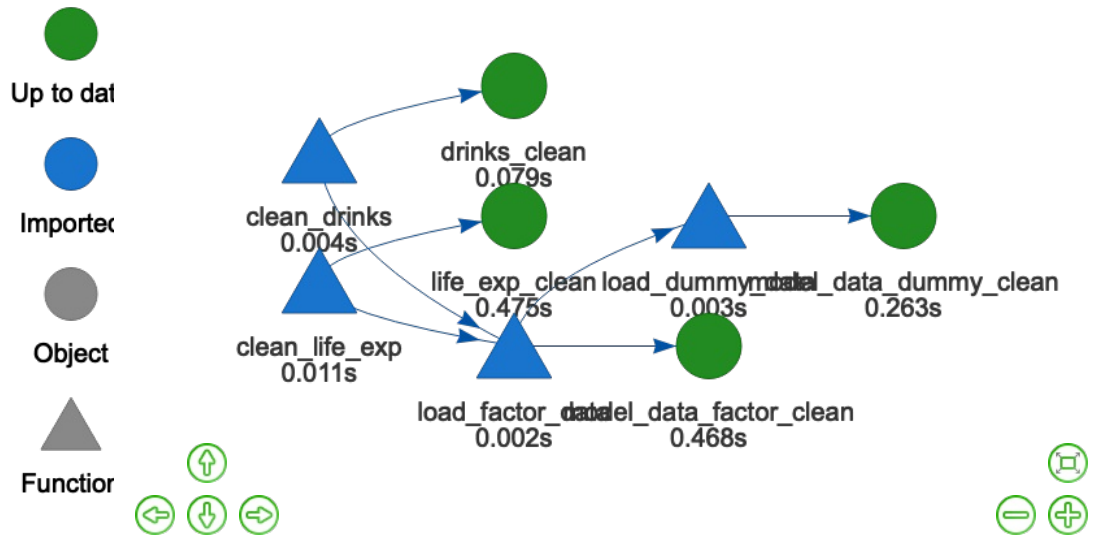
```
## All targets are already up to date.
```

Visualize dependency structure

After defining our data pipeline, we can visualize the dependency structure in a directed graph just like we did with our basic example.

```
dcw_config <- drake_config(dcw_plan)
vis_drake_graph(dcw_config)
```

Dependency graph



Just like in the other example, if we were to update any part of the pipeline upstream, Drake would automatically update the dependent parts of the pipeline by running the make() function again.