gnumaker

Version: 0.0.0.9008

Overview

gnumaker makes if easy to create and use GNU Makefiles to aid a reproducible work flow for data analysis projects.

GNU Make is the defacto standard for efficiently rerunning appropriate steps in the data analysis or reporting process if a particular file is changed. Only the necessary steps are rerun.

Rather than creating a new system for setting up and building output from statistical software syntax files, **gnumaker** leverages off existing GNU Make rules. These rules, for R, Sweave, R Markdown, Stata, SAS and other syntax files are available at r-makefile-definitions on Github. These are described in P Baker (2020) Using GNU Make to Manage the Workflow of Data Analysis Projects, *Journal of Statistical Software (Accepted)*.

For those not familiar with GNU Make, **gnumaker** allows simple dependencies between files to be specified to produce a working Makefile and the associated directed acyclic graph (DAG). I'd welcome Github issues containing error reports or feature requests. Alternatively, you can email the package maintainer at drpetebaker at gmail dot com.

Installation

You can install the development version of **gnumaker** from GitHub with:

```
## if you don't have devtools installed, automatically install it from CRAN
if (!requireNamespace("devtools", quietly = TRUE))
   install.packages("devtools")
devtools::install_github("petebaker/gnumaker")
```

Usage

There are four key functions in **gnumaker**. These are:

- `create_makefile()` creates a gnu_makefile object given dependencies between syntax, data and output files,
- `write_makefile()` writes a Makfile to disk,
- `info_rules()` provides information about data analysis GNU Make rules for various target and dependency filename extensions, and
- `plot()` plots a DAG for a gnu_makefile object.

Example

Suppose we have a data file `simple.csv` and use `read.R` to read and clean the data. After storing the cleaned data in a .RData file, we then employ `linmod.R` to plot and analyse the data. Next, using the stored results, two reports `report1.pdf` and `report2.docx` are produced from `report1.Rmd` and `report2.Rmd`. The workflow may be encapsulated in a Makefile which is then employed to manage the process and generate or regenerate any intermediate files when the data or syntax changes.

Using the **gnumaker** package we simply need to provide a list of targets to the the `create_makefile` function where the components specify a target as a name and dependency file(s) as a character vector. The package uses the GNU Make pattern rules in *r-rules.mk* to choose file names for targets but we can override the defaults.

For instance, in this example the first two dependency files are `simple.csv` and `read.R` so we provide the first target as the first component of the list as `read = c("read.R", "simple.csv")`, where the name `read`

can be anything we like. The second target depends on the `read` target and `linmod.R` and so we specify this with `linmod = c("linmod.R", "read")` and so on.

Target file names are substituted using defaults and the *Makefile* is rearranged using the DAG of the relationships. For instance, the default target file for the first dependency in the `read` component, which is `read.R`, becomes `read.Rout` but we can change the default target file extension for all `.R` files using the `default.exts` argument and specify say a HTML target file with `default.exts = list(R = "html")`.

Finally we specify the first target (usually `all`) as two reports `report1.pdf` and `report2.docx` using `target.all = c("rep1","rep2")` which by default would be `report1.html` and `report2.html` but which we specify as `report1.pdf` and `report2.docx` by specifying the option `all.exts = list(rep1 = "pdf", rep2 = "docx")`.

To run all R script filesand analyses in order we simply type `make` in a terminal or set up `RStudio` or our IDE to use GNU Make as the build mechanism which allows us to (re)run analyses by pressing the appropriate Build button.

The `Makefile` is specified, printed and plotted using:

```
library(gnumaker)
gm1 <-
  create_makefile(targets = list(read = c("read.R", "simple.csv"),
                  linmod = c("linmod.R", "read"),
                  rep1 = c("report1.Rmd", "linmod"),
                  rep2 = c("report2.Rmd", "linmod")),
                  target.all = c("rep1", "rep2"),
                  all.exts = list(rep1 = "pdf", rep2 = "docx"),
 comments = list(linmod = "plots and analysis using 'linmod.R'"))
```

A Makefile `Makefile.demo` is produced with `write_makefile(gm1)`

```
write_makefile(gm1, file = "Makefile.demo")
#> File: Makefile.demo written at Fri Jul 10 19:12:13 2020
```

```
# File: Makefile.demo
# Created at: Fri Jul 10 19:12:13 2020

# Produced by gnumaker:  0.0.0.9008 on R version 3.6.3 (2020-02-29)
# Before running make, please check file and edit if necessary

# .PHONY all target which is run when make is invoked
.PHONY: all
all: report1.pdf report2.docx

# report1.pdf depends on report1.Rmd, linmod.Rout
report1.pdf: report1.Rmd linmod.Rout

# report2.docx depends on report2.Rmd, linmod.Rout
report2.docx: report2.Rmd linmod.Rout

# plots and analysis using 'linmod.R'
linmod.Rout: linmod.R read.Rout

# read.Rout depends on read.R, simple.csv
read.Rout: read.R simple.csv

# include GNU Makfile rules. Most recent version available at
```

```
# https://github.com/petebaker/r-makefile-definitions
include ~/lib/r-rules.mk

# remove all target, output and extraneous files
.PHONY: cleanall
cleanall:
    rm -f *~ *.Rout *.RData *.docx *.pdf *.html *-syntax.R *.RData
```

The DAG of the `gnu_makefile` object can be produced with `plot(gm1)`.

```
plot(gm1)
```

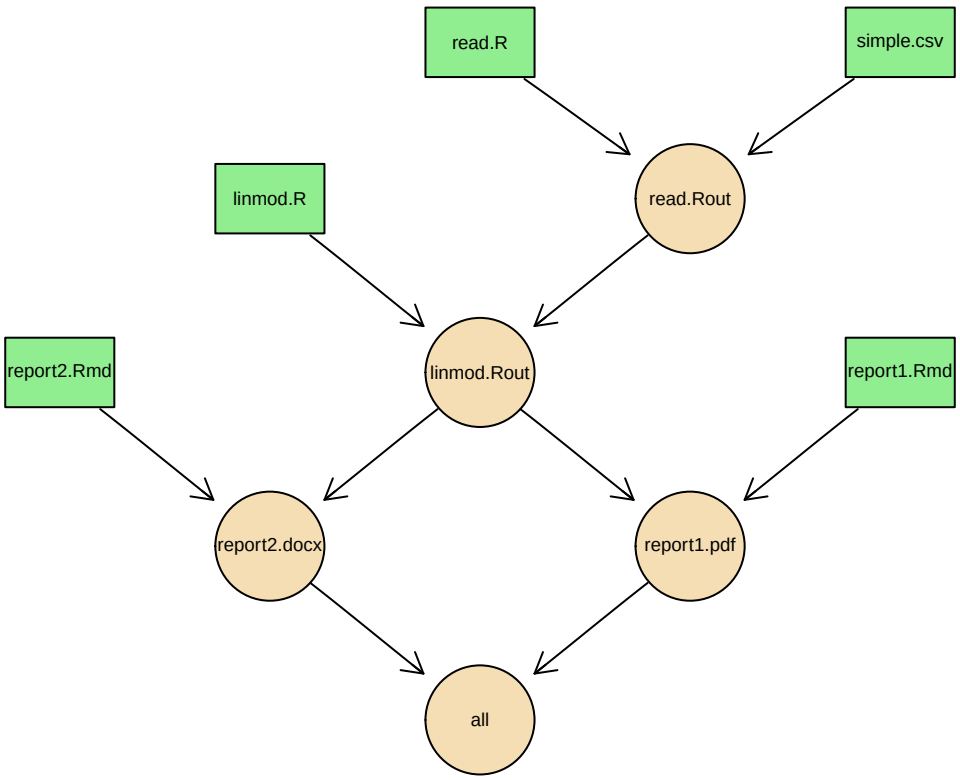

Figure 1: DAG of Makefile for simple example. The DAG of the `gnu_makefile` object can be produced with `plot(gm1)`. Using the minimal set of files (shown in green rectangles), then GNU Make allows us to (re)generate all other files shown as wheat coloured circles)

We can use the function `info_rules` to determine the possible target files for dependency files. For instance, what target files have `Makefile` rules for an `.R` R syntax file?

```
info_rules("R")
#> Possible filename extensions for 'R':
#> [1] "docx" "html" "odt"  "pdf"  "Rout" "rtf"
#>
#> Default: 'Rout'
#>
#> Example rules:
#> example1.Rout: example1.R dep_file2 dep_file3
#>  or
#> example1.Rout: {@:.Rout=.R} dep_file2 dep_file3
#>
```

```
#> NB: For further help on Makefile rules, type 'make help' in a terminal once
#>     an appropriate 'Makefile' is present in the current directory
```

For `.Rmd` R Markdown files, use

```
info_rules("Rmd")
#> Possible filename extensions for 'Rmd':
#>  [1] "_beamer-handout.Rmd" "_beamer.pdf"         "_ioslides.html"
#>  [4] "_slidy.html"         "_tufte.pdf"          "-syntax.R"
#>  [7] "docx"                "html"                "odt"
#> [10] "pdf"                 "pptx"                "rtf"
#>
#> Default: 'html'
#>
#> Example rules:
#> example1.html: example1.Rmd dep_file2 dep_file3
#>  or
#> example1.html: {@:.html=.Rmd} dep_file2 dep_file3
#>
#> Other options are available for R Markdown files, such as:
#>
#> example1_ioslides.html: example1.Rmd dep_file2 dep_file3
#> example1_beamer.pdf: example1.Rmd dep_file2 dep_file3
#>
#> to produce ioslide and beamer presentation formats.
#>
#> An R syntax file can be produced with
#>   make example1-syntax.R
#> and a similar rule can be specified if required with
#> example1-syntax.R: example1.Rmd dep_file2 dep_file3
#>
#> NB: For further help on Makefile rules, type 'make help' in a terminal once
#>     an appropriate 'Makefile' is present in the current directory
```

For more examples, see the gnumaker vignette (under construction).

Notes

gnumaker is under construction and could change (and improve) rapidly at various times but this depends on work/life balance.

To do

- DONE extract dependency and target file extensions in `r-rules.mk`, preferably by parsing the included file (done using `pattern-exts`')
- DONE incorporate dependency and target file extensions extracted using `pattern-exts` into `create_makefile` and set defaults
- STARTED move `pattern_exts` to internal functions and create `show_extensions` to assist user specification
- TODO allow specification of *global options* in `zzz.R` so that it is easier to customise defaults e.g. so user can specify defaults in .Rprofile
- TODO add `testthat` unit testing for more complicated examples
- TODO add travis.ci and other automatic checking - see r-pkgs.had.co.nz/release.html
- TODO allow for target file extensions and dependency files to be set as user specified variables which would make the `Makefile`s produced more flexible but less easy to read

- TODO allow `target.all` to be determined from DAG if this is sensible
- TODO either incorporate `makefile2graph` as a way of plotting `Makefile`s not made with **gnumaker** or write own functions. (See makefile2graph on github)