rmake: A Build Process Manager for Complex Analyses in R

Michal Burda
Institute for Research and Applications of Fuzzy Modeling
Centre of Excellence IT4Innovations, Division University of Ostrava
30. dubna 22, 701 03 Ostrava, Czech Republic
E-mail: `Michal.Burda@osu.cz`

Abstract

R software allows to develop *repeatable* statistical analyses, i.e., automatically re-computable analyses if data or some data processing step changes. However, if the analyses grow on complexity, their manual re-execution on any change may become tedious and prone to errors. *Make* is a widely accepted tool for managing the generation of resulting files from source data and script files. *Make* reads dependencies between data and script files from a text file called the `Makefile`. The aim of this paper is to present **rmake**, an R package for easy generation of `Makefile`s for statistical and data manipulation projects.

Keywords: statistical analyses, build process, `make`, `Makefile`, R.

1. Introduction

R (R Core Team 2017) is a mature scripting language for statistical computations and data processing. Besides other benefits, an important advantage of R is that it allows to write *repeatable* statistical analyses, that is, to program all steps of data processing in a scripting language, which allows to re-execute the whole process after any change in data or in any processing step.

There are several useful packages for R to obtain repeatability of statistical computations much more easier. Among others, let us name **knitr** (Xie 2015, 2017) and **rmarkdown** (Allaire, Xie, McPherson, Luraschi, Ushey, Atkins, Wickham, Cheng, and Chang 2017). These tools allow to write R scripts that generate reports combining text with tables and figures generated from data. Creation of final statistical reports by such scripts is as simple as issuing a single statement from the command line or as clicking an icon in an integrated development environment (IDE) such as *RStudio* (RStudio Team 2015).

However, if the analyses grow on complexity, manual re-execution of the whole process may become tedious, prone to errors, and very demanding from the view of computational power. Complex analyses involve typically a lot of pre-processing steps on large data sets, a repetitive execution of commands differing in several parameters only, and producing multiple output files in various formats. It is very inefficient to re-run over and over again all the pre-processing steps to refresh the final report after any change on it. A caching mechanism provided by **knitr** is very helpful there, but its use is still limited on a single report. It is rational to split a complex analysis into several parts and save intermediate results into files. However, such approach brings another challenge: management of dependencies between inputs, outputs and underlying scripts to ensure a refresh of all the results on change of any prerequisite. In open source community, (GNU) *Make* is a popular tool to help with that.

Make is a tool which controls the generation of intermediate and resulting files from source data and script files. It was primarily created to help program developers to build executable binaries from source codes. However, it can be used to generate any type of file. *Make* gets its knowledge of how to build the results from a file called the `Makefile`, which defines dependencies between files as well as commands of how to create dependent files from their sources. *Make* compares last-change timestamps of the files listed in `Makefile` to determine which files (and in what order) have to be refreshed in order to get all of them updated.

Make is also quite popular in the R community, with a direct support e.g. by *RStudio* and another tools. It is quite straightforward to write `Makefile` manually and many R users write simple `Makefile`s for themselves by hand. However, for more complex build processes, manual management of `Makefile` may become grueling. The aim of this paper is to present **rmake**, an R package providing tools for easy generation of `Makefile`s for statistical and data manipulation tasks in R. The main features of **rmake** are as follows:

- the use of the well-known *Make* tool;
- easy definitions of file dependencies in the R language;
- high flexibility provided by parameterized execution of R scripts and programmatically generated dependencies;
- simple and short code thanks to special `%>>%` pipeline operator and templating mechanism;
- support for R scripts and R markdown files;
- extensibility for user-defined rule types;
- isolated and parallel execution of building tasks obtained for free thanks to *Make*'s parallel processing features;
- support for all platforms including Unix (Linux), MacOS, MS Windows, and Solaris;
- compatibility with *RStudio*.

A different approach to build complex analytical projects is represented e.g. by the **drake** package (Landau 2018), which (unlike to **rmake**) processes all the tasks within the single R session and which handles all the dependencies between R objects by itself. To the opposite, **rmake** is simply a light-weight generator of the `Makefile` dependency file, which leaves the re-generation of obsolete results on the *Make* utility. A large list of other pipelining projects may be found also at `https://github.com/pditommaso/awesome-pipeline`.

The rest of the paper is organized as follows. Section 2 lists all necessary steps for correct installation and setup. Section 3 describes the basic usage of the **rmake** package including initialization of a new project, creating the build rules, running the build process, and visualization of file dependencies. Section 4 provides all details about pre-defined build rules and custom rules definitions. In Section 5, the advanced topics of **rmake** usage are discussed: the mechanism of tasks for grouping the rules that have to be executed together, parameterized execution of rules, and rule templates. Section 6 concludes the paper.

2. Installation

In order to use **rmake**, the R environment and the *Make* program has to be installed and properly configured, in advance. On Linux-based systems, it is usually a matter of installing the appropriate distribution packages. On Windows, an installation of `Rtools` is recommended that contains the *Make* tool included.

To install the **rmake** package from CRAN, execute the following command from within the R session (note the leading "`R>`" denotes the R's shell prompt and it is not a part of the command):

```
R> install.packages("rmake")
```

Alternatively, a development version of **rmake** may be obtained directly from GitHub:

```
R> install.packages("devtools")
R> devtools::install_github("beerda/rmake")
```

2.1. Settings for Use Outside of the R Session

For **rmake** to work properly, `R_HOME` and `R_ARCH` environment variables have to be set correctly. If executing *Make* from within an R session (e.g., from *RStudio*), the environment variables are set automatically by R. In order to execute *Make* outside R, for instance, from the shell, these variables have to be set manually. The `R_HOME` variable should contain a path to R's installation directory and `R_ARCH` the architecture variant. The correct values may be obtained from R by issuing the following commands:

```
R> Sys.getenv("R_HOME")

[1] "/usr/lib/R"

R> Sys.getenv("R_ARCH")

[1] ""
```

To set environment variables on a Unix-like system, issue the following shell command ("`$`" is a shell prompt):

```
$ export R_HOME=/usr/lib/R
$ export R_ARCH=
```

These commands may be added to your home `.profile` file for variables to be created automatically after you log-in.

3. Basic Usage

3.1. New Project Initialization

To start maintaining the R project with **rmake**, an R script `Makefile.R` has to be created that would then generate the `Makefile`. That script file may be created manually, or from a skeleton provided by **rmake**. To start from skeleton, first load the **rmake** package:

```
R> library(rmake)
```

and then enter the following command:

```
R> rmakeSkeleton(".")
```

This will create two files in the current directory ("`.`"): `Makefile.R` and `Makefile`. The first file is an R script intended to generate the second file. In the beginning, `Makefile.R` contains the following:

```
library(rmake)
job <- list()
makefile(job, "Makefile")
```

Function `makefile()` generates `Makefile` based on a `job` variable, which currently contains an empty list. Nevertheless, the generated `Makefile` contains at least a single rule that ensures an automatic re-creation of `Makefile` after any change to the `Makefile.R` script is made in the future.

3.2. Running the Build Process

Once the `Makefile` exists, the *Make* tool may be executed from within the R session by calling the following function:

```
R> make()
```

```
make: Nothing to be done for 'all'.
```

Indeed, nothing is to be done, since the single rule generating the `Makefile` itself needs not to be re-generated. After `Makefile.R` gets updated, `Makefile` would be re-generated and also other tasks will be executed, as specified by the rules in `Makefile.R`.

To run *Make* from shell, just enter the following command (note the settings needed for everything to work properly outside the R session in Section 2.1):

```
$ make

make: Nothing to be done for 'all'.
```

If working in *RStudio*, it is beneficial to setup its environment to use Make: in *Build/Configure Build Tools* menu, set *Project build tools* to *Makefile*. A *Build All* command becomes available that runs *Make* using the generated `Makefile`.

3.3. Adding a Build Rule

Now, let us do some “real” work. Suppose we have `data.csv` with the following content:

```
ID,V1,V2
a,2,8
b,9,1
c,3,3
```

We would like to compute the sums of columns `V1` and `V2` and store the result into a file `sums.csv`. Therefore, we create the following script file `script.R`:

```r
d <- read.csv("data.csv")
sums <- data.frame(ID="sum",
                   V1=sum(d$V1),
                   V2=sum(d$V2))
write.csv(sums, "sums.csv", row.names=FALSE)
```

The script reads `data.csv` into a variable `d`, creates a data frame `sums` with computed sums and writes it to the file `sums.csv`.

Now, let us modify the `Makefile.R` script to build the `sums.csv` file automatically whenever either `data.csv` or `script.R` files change. All that has to be done is to update the line of code where the `job` variable is created in `Makefile.R`:

```r
library(rmake)
job <- list(rRule(target="sums.csv", script="script.R", depends="data.csv"))
makefile(job, "makefile")
```

Function `rRule()` creates a new rule for execution of an R script `script.R`, whose target is `sums.csv` and which depends on `data.csv`. Whenever any dependency file or the script file changes, the rule triggers and re-executes the given script in order to build the given target. Let us run the `make` command again:

```r
R> make()
```

The *Make* utility should firstly re-generate the `Makefile` itself (since `Makefile.R` has changed) and then execute `script.R` in a new R session to create `sums.csv`. Further calls of `make` will do nothing until any change is detected again.

To finalize this toy example, let us create a file named `analysis.Rmd` with the following content:

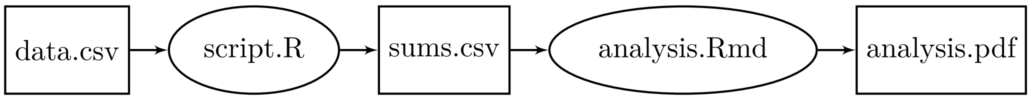

Figure 1: A simple chain of dependencies from the example in Section 3.3

~~~
---
title: "Analysis"
output: pdf_document
---

# Sums of data rows

```{r, echo=FALSE, results='asis'}
sums <- read.csv('sums.csv')
knitr::kable(sums)
```
~~~

This is a markdown file, which we are going to process with the **rmarkdown** package to create a PDF document with the results. This script reads `sums.csv` and prints its content in a tabular layout. For more details on how to work with markdown documents see Allaire *et al.* (2017).

The `analysis.Rmd` script depends on the `sums.csv` data file. The markdown processor produces an `analysis.pdf` file from it. Let us now update `Makefile.R` so that the PDF file is refreshed everytime either the script or the data change. The job creation command should be modified as follows:

```r
library(rmake)
job <- list(
  rRule(target="sums.csv", script="script.R", depends="data.csv"),
  markdownRule(target="analysis.pdf", script="analysis.Rmd",
               depends="sums.csv")
)
makefile(job, "makefile")
```

See Fig. 1 for an illustrative diagram of dependencies. After calling

```r
R> make()
```

the `analysis.pdf` is generated.

3.4. The Pipe Operator

The sequence of the above-listed **rmake** rules makes a chain: `data.csv` is a prerequisite for `sums.csv` which is a prerequisite for `analysis.pdf`. Such sequence of rules may be equivalently written using the new "`%>>%`" pipe operator introduced by **rmake** (cf. with Fig. 1):

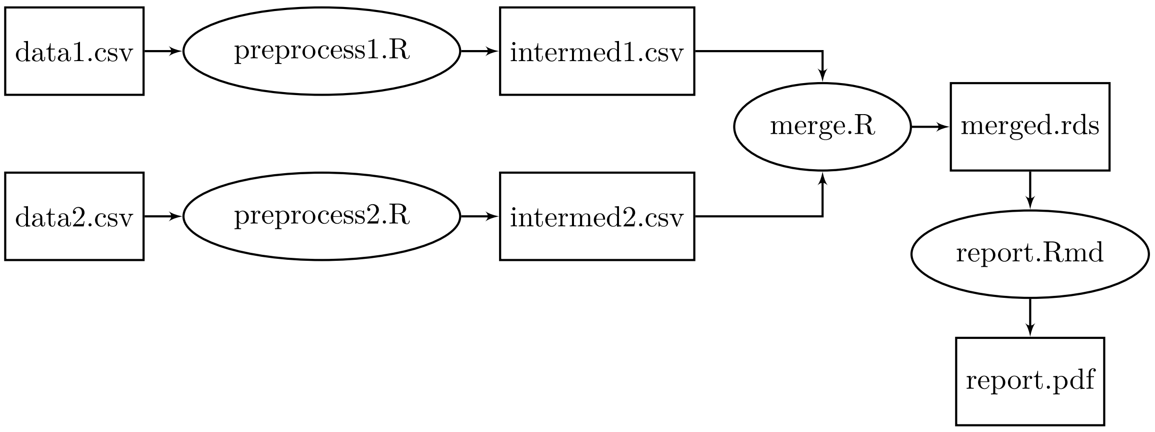

Figure 2: More complex chain of dependencies from the example in Section 3.4

```
R> job <- "data.csv" %>>% rRule("script.R") %>>%
+    "sums.csv" %>>% markdownRule("analysis.Rmd") %>>%
+    "analysis.pdf"
```

Generally, every k-th element of the pipe chain (for $k = 2, 4, 6, \ldots$) must be a call to a function that creates a **rmake** rule. Prior its execution, each such function call internally obtains two additional named arguments: `depends` and `target`, whose values are respectively obtained from the preceding (i.e., $(k-1)$-th) or following (i.e., $(k+1)$-th) element of the chain.

If some rule depends on or creates multiple files, their names have to be specified as a character vector (see the `c()` function) – for instance, the `run.R` script reads and writes two files:

```
R> job <- c('in1.csv', 'in2.csv') %>>%
+    rRule('run.R') %>>%
+    c('out1.csv', 'out2.csv')
```

If the dependencies are more complex than a single chain, multiple chains may be merged with the `c()` function as follows:

```
R> chain1 <- "data1.csv" %>>% rRule("preprocess1.R") %>>% "intermed1.rds"
R> chain2 <- "data2.csv" %>>% rRule("preprocess2.R") %>>% "intermed2.rds"
R> chain3 <- c("intermed1.rds", "intermed2.rds") %>>% rRule("merge.R") %>>%
+    "merged.rds" %>>% markdownRule("report.Rmd") %>>% "report.pdf"
R>
R> job <- c(chain1, chain2, chain3)
```

A graphical representation of defined dependencies is shown in Fig. 2.

3.5. Cleaning Up

A good manner of `Makefile` writers is to provide a clean-up task that deletes all files of the project that were generated within the build process. This is traditionally executed by the following command:

```
$ make clean
```

Each **rmake**'s rule type adds automatically to the `Makefile` a command for deleting all target files that were generated by that rule. A single exception is the `Makefile` itself that is never deleted, although it is generated too.

From within R, the clean-up task may be executed by

```
R> make("clean")
```

3.6. Parallel Execution

Some implementations of the *Make* utility allow to build multiple targets in parallel. For instance, *GNU Make* recognizes the `-j` argument, which can be used to specify the number of processes to run simultaneously. For instance,

```
$ make -j8
```

causes up to 8 targets to be prepared in parallel. If the `-j` option is given without a number, the *Make* utility will not limit the number of rules that can run simultaneously.

From R, the parallel execution might be started with the following command:

```
R> make("-j8")
```

3.7. Visualization

The list of rules may be printed, to see concisely the defined dependencies. For instance, the `job` from Section 3.4 would produce the following output:

```
R> print(job)

[[1]]
(preprocess1.R, data1.csv) -> R -> (intermed1.rds)
[[2]]
(preprocess2.R, data2.csv) -> R -> (intermed2.rds)
[[3]]
(merge.R, intermed1.rds, intermed2.rds) -> R -> (merged.rds)
[[4]]
(report.Rmd, merged.rds) -> markdown -> (report.pdf)
```

A lot more comprehensible view on the graph of dependencies is obtained by visualizing the `job`:

```
R> visualize(job, legend=FALSE)
```

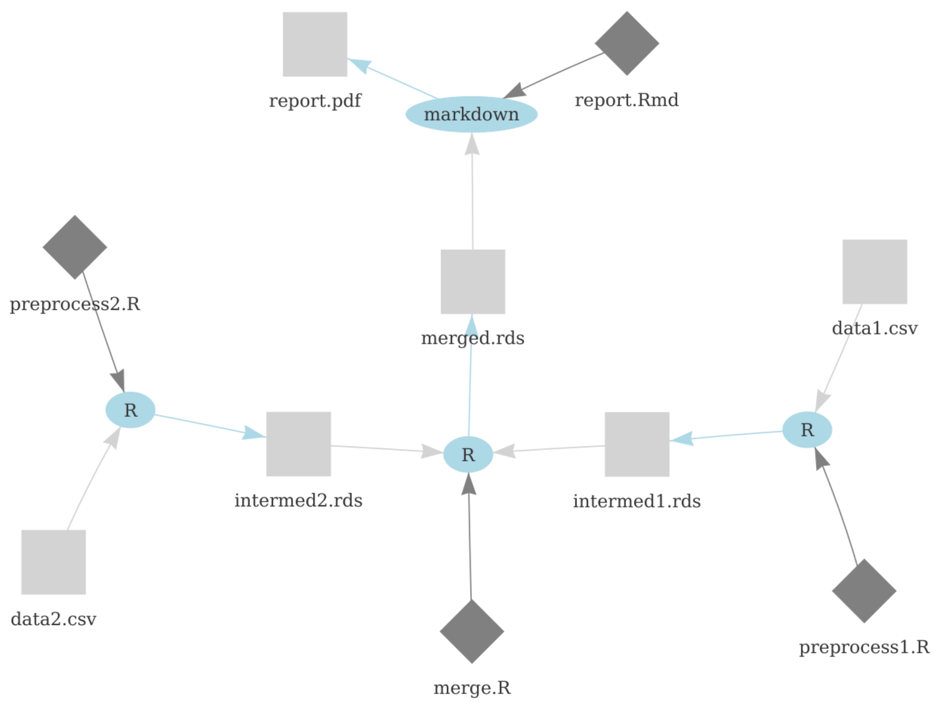

Figure 3: Visualization of the `job` from Section 3.4 with the `visualize()` function

The `visualize()` function converts the `job` into a **visNetwork**'s directed graph and renders it as an interactive web-page, in which the picture may be zoomed and nodes re-arranged with a mouse – see Fig. 3 for an overview. The optional `legend` argument turns off the legend in the resulting figure.

In the graph, the main script files are depicted with diamonds, rules are represented with ovals, while the other files are symbolized with squares.

4. Details on Build Rules

The **rmake** package provides several functions that represent the most common build-rule types. Each function has a mandatory `target` argument for a character vector of file names created by that rule. Additionally, a character vector of file names the rule depends on can be specified as an optional `depends` argument. The `task` argument allows rules to group into *tasks* – see Section 5.1 for more details. Some rules also allow an optional `params` argument to pass parameters to the scripts; Section 5.2 contains detailed information on that topic. Each rule, when executed by the *Make* tool, is started as a separate operating system process, that is, the R scripts and markdown processors do not share the running instance of the R interpreter.

4.1. Pre-defined Rule Types

```
rRule(target, script, depends = NULL, params = list(), task = "all")
```

The `rRule()` rule executes the R script by calling the `Rscript` executable file from the shell and `source`-ing the `script` file. This rule is fired if either any file from `depends` or the `script` file changes. Cleaning removes all `target` files.

```
markdownRule(target, script, depends = NULL, format = "all",
             params = list(), task = "all")
```

The `markdownRule()` rule renders a `target` document from the source Markdown `script` file. The rendering is done by calling the `render()` function of the **rmarkdown** package (Allaire *et al.* 2017).

The `format` argument is passed to the **rmarkdown**'s `render()` function as an `output_format` parameter and thus may be used to specify the desired format of the resulting file: `'all'` to render all formats defined by the **rmarkdown**'s output directive specified in the `Rmd` file, a name of the format to render a single format or a vector of format names to render multiple formats. The recognized format names are: `'html_document'` (HTML web page), `'pdf_document'` (Adobe Portable Document Format), `'word_document'` (Microsoft Word document), `'odt_document'` (OpenDocument Text), `'rtf_document'` (Rich Text Format), and `'md_document'` (Markdown document).

Cleaning removes all `target` files.

```
offlineRule(target, message, depends = NULL, task = "all")
```

The `offlineRule()` rule provides a way to force some non-automated action within the *Make* build process. It should be used whenever a transformation of prerequisites from the `depends` vector into `target` files requires a manual action. In case of building, a custom error `message` is shown that would instruct the user to perform the task by hand until the `target` files get more recent modification timestamps than files in `depends` vector. Cleaning removes all `target` files.

4.2. Custom Rules

Besides predefined rule types, any custom rule may be added to the build process. Internally, the `makefile()` function generates `Makefile` from a list of instances of the S3 `rmake.rule` class. To create an object of such type, the following general function may be used:

```
rule(target, depends = NULL, build = NULL, clean = NULL,
     task = "all", phony = FALSE)
```

where `target` is a character vector of target files that the rules is intended to generate, and `depends` is a character vector of prerequisite file names. To the contrast with predefined rule types, custom rules should add also script file names into a vector of dependencies, preceding other data files.

The `task` argument may be used to assign the rule to a certain task, see Section 5.1 for more information.

The `phony` argument is a boolean (`TRUE`/`FALSE`) value specifying whether the *Make* rule has a `.PHONY` (i.e. non-file) target. A rule should be marked with `phony` if the target is not a file name that would be generated by the `build` command. E.g., `all` or `clean` are phony targets. Also all targets representing tasks (see Section 5.1) are phony. See also the manual to the *Make* tool for more details on `.PHONY` targets.

The `build` and `clean` arguments are character vectors with shell commands that have to be executed during the build or cleaning-up, respectively. It is advisable to write shell commands with the use of *Make* variables that are predefined by **rmake** at the beginning of the `Makefile`:

- `$(R)` – a path and file name of the `Rscript` binary, as obtained from the `R_HOME` and `R_ARCH` environment variable (see Section 2.1 for details);
- `$(RM)` – a name of the file deletion command (`rm` on Unix, `del` on Windows).

For instance, the following rule runs *NodeJS* JavaScript interpreter on `test.js` script, which generates `test.json` file:

```
R> r <- rule(target="test.json", depends="test.js", build="node test.js",
+       clean="$(RM) test.json")
```

Make variables other than `$(R)` or `$(RM)` may be defined by modifying the `defaultVars` global variable, e.g., let us introduce a new `$(JS)` variable with path to the JavaScript interpreter:

```
defaultVars["JS"] <- "/usr/bin/node"
```

One can then modify the above-listed rule in `Makefile.R` to use the new `$(JS)` variable:

```
library(rmake)
defaultVars["JS"] <- "/usr/bin/node"
job <- list(rule(target="test.json",
                 depends="test.js",
                 build="$(JS) test.js",
                 clean="$(RM) test.json"))
makefile(job, "Makefile")
```

The **rmake** package provides a useful tool to help write rules that execute custom R sequence of commands. The `inShell()` function simply takes an R language expression and transforms it into a character vector with a shell command that calls `Rscript` with parameters that execute the given expression:

```
R> inShell({ result <- 1+1; saveRDS(result, "result.rds") })

[1] "$(R) -e '{' \\"
[2] "-e '    result <- 1 + 1' \\"
[3] "-e '    saveRDS(result, \"result.rds\")' \\"
[4] "-e '}'"
```

The `inShell()` function may be used to define a new rule:

```
rule(target="result.rds",
     build=inShell({ result <- 1+1; saveRDS(result, "result.rds") }),
     clean="$(RM) result.rds")
```

However, the overuse of `inShell()` is not recommended. For normal processing of data, it is far better to store R commands into a separate script file, because then, after any change to the code is made, the *Make* tool can detect it and force update of other depending artifacts. Any rule change at the level of `Makefile` will not cause any rebuild of the target. The intended use of `inShell()` is to ease the implementation of internals such as in `rRule()` or `markdownRule()`.

5. Advanced Usage

5.1. Tasks

The mechanism of tasks allows to make groups of rules. Groups of rules may then be executed together. If not stating differently, each **rmake** rule is a member of the `all` task. **rmake** also provides a special `clean` task for project cleanup. To build all rules grouped into a task, simply invoke the `make` command and give it the name of the task. For instance:

```
$ make all
```

executes all rules grouped into the `all` task. Equivalently, we may execute the task from within the R session:

```
R> make('all')
```

A rule is assigned to a task by specifying task name in the `task` argument of the rule creation function. A rule may be a member of more than a single task – simply put all task names in the character vector as the `task` argument.

In the following example, the rules are divided into two tasks:

```
library(rmake)
job <- c(
  "data.csv" %>>% rRule("preprocess.R") %>>% "data.rds",
  "data.rds" %>>% markdownRule("preview.Rmd", task="preview") %>>%
      "preview.pdf",
  "data.rds" %>>% markdownRule("final.Rmd", task="final") %>>% "final.pdf"
)
makefile(job, "Makefile")
```

This code creates a rule `preprocess.R`, which transforms `data.csv` into `data.rds`, and two markdown rules, `preview.Rmd` and `final.Rmd`, that are each assigned to its own task named `preview` and `final`, respectively. Thus invoking

```
R> make("preview")
```

would create `data.rds` (because `preview.Rmd` depends on it, irrespective of `preprocess.R` is a member of the `all` task) and `preview.pdf`, but not `final.pdf`. Similarly,

```
R> make("final")
```

would generate `final.pdf` (possibly with previous build of `data.rds`, if it was not done already), but not `preview.pdf`. All rules will be triggered by calling

```
R> make("all")
```

5.2. Parameterized Execution of Rules

The **rmake** package allows to send parameters to the main script of a rule. Both `rRule()` and `markdownRule()` functions may obtain a list of arbitrary data as the `params` argument. The content of that argument would be available as the `params` global variable from within the script. Through such parameterization, a single R script may be used to produce multiple outputs. An example is as follows:

```
library(rmake)
job <- c(
  "data.csv" %>>% rRule("fit.R", params=list(alpha=0.1)) %>>% "out-0.1.rds",
  "data.csv" %>>% rRule("fit.R", params=list(alpha=0.2)) %>>% "out-0.2.rds",
  "data.csv" %>>% rRule("fit.R", params=list(alpha=0.3)) %>>% "out-0.3.rds",
  "data.csv" %>>% rRule("fit.R", params=list(alpha=0.4)) %>>% "out-0.4.rds"
)
makefile(job, "Makefile")
```

We can create the following `fit.R` script file to see what is inside of the `params` global variable:

```
# the fit.R file
str(params)
```

Executing

```
R> make("all")
```

will show the following result:

```
List of 5
 $ .target : chr "out-0.1.rds"
 $ .script : chr "fit.R"
 $ .depends: chr "data.csv"
 $ .task    : chr "all"
 $ alpha    : num 0.1

List of 5
 $ .target : chr "out-0.2.rds"
 $ .script : chr "fit.R"
 $ .depends: chr "data.csv"
 $ .task   : chr "all"
 $ alpha   : num 0.2

List of 5
 $ .target : chr "out-0.3.rds"
 $ .script : chr "fit.R"
 $ .depends: chr "data.csv"
 $ .task   : chr "all"
 $ alpha   : num 0.3

List of 5
 $ .target : chr "out-0.4.rds"
 $ .script : chr "fit.R"
 $ .depends: chr "data.csv"
 $ .task   : chr "all"
 $ alpha   : num 0.4
```

Indeed, the `params` variable contains the `alpha` parameter with an expected value. Besides that, `params` contains several dot-named values that correspond to the arguments of `rRule()`: `.target`, `.script`, `.depends`, and `.task`.

The `fit.R` script may handle the `params` global variable directly, but it is advisable to use the `getParam()` function instead, which throws an error in case the script is executed without `params` being defined before:

```r
# the fit.R file
library(rmake)

dataName <- getParam(".depends")
resultName <- getParam(".target")
alpha <- getParam("alpha")

# now we can use these variables to do here some real work...

cat("dataName:", dataName, "\n")
cat("resultName:", resultName, "\n")
cat("alpha:", alpha, "\n")
```

Executing `fit.R` outside the generated `Makefile` would trigger a warning message about non-existence of the parameters:

```
R> source("fit.R")
```

```
dataName: NA
resultName: NA
alpha: NA
Warning message:
In getParam(".depends") :
  rmake parameters not found - using default value for ".depends": NA
Warning message:
In getParam(".target") :
  rmake parameters not found - using default value for ".target": NA
Warning message:
In getParam("alpha") :
  rmake parameters not found - using default value for "alpha": NA
```

Meaningful default values may be assigned to the parameters by a second argument of the `getParam()` function:

```
dataName <- getParam(".depends", "data.csv")
resultName <- getParam(".target", "result.rds")
alpha <- getParam("alpha", 0.2)
```

The warning does not disappear, but the script has now a chance to run with proper parameters, which may be useful, when debugging the script in *RStudio*:

```
R> source("fit.R")
```

```
dataName: data.csv
resultName: result.rds
alpha: 0.2
Warning message:
In getParam(".depends", "data.csv") :
  rmake parameters not found - using default value for ".depends": data.csv
Warning message:
In getParam(".target", "result.rds") :
  rmake parameters not found - using default value for ".target": result.rds
Warning message:
In getParam("alpha", 0.2) :
  rmake parameters not found - using default value for "alpha": 0.2
```

5.3. Rule Templates

More complex analyzes may contain similar rule sequences that repeate multiple times. Think of fitting multiple models differing only in some parameters, stored into files with a name derived from parameter values. The **rmake** package provides a templating mechanism to avoid tedious copy&paste of rule definitions and to help quickly creating and easily maintaining the `Makefile.R` script.

The idea of rule templates is best presented on the following example. Let us have a lot of CSV data files that all have to be processed and saved in a uniform way. We may create a script that processes all files in a loop, but that would make difficult a selective re-calculation of future changed data. Instead, we might to write a unique **rmake** rule for each data file:

```
R> job <- c(
+    "data-1.csv" %>>% rRule("process.R") %>>% "result-1.csv",
+    "data-2.csv" %>>% rRule("process.R") %>>% "result-2.csv",
+    # ...
+    "data-99.csv" %>>% rRule("process.R") %>>% "result-99.csv"
+  )
```

Instead of that, rule templates will simplify the code significantly:

```
R> tmpl <- "data-$[NUM].csv" %>>% rRule("process.R") %>>% "result-$[NUM].csv"
R> variants <- data.frame(NUM=1:99)
R> job <- expandTemplate(tmpl, variants)
```

The `expandTemplate()` function simply takes a list of rules, `tmpl`, and replaces all appearances of *template variables* in all strings with their values provided by the `variants` data frame. The rules in `tmpl` are replicated for each row of the `variants` data frame.

The following example creates rules for each combination of `DATA` and `TYPE`:

```
R> variants <- expand.grid(DATA=c("dataSimple", "dataComplex"),
+                          TYPE=c("lm", "rf", "nnet"))
R> print(variants)

         DATA TYPE
1  dataSimple   lm
2 dataComplex   lm
3  dataSimple   rf
4 dataComplex   rf
5  dataSimple nnet
6 dataComplex nnet

R> tmpl <- "$[DATA].csv" %>>% rRule("fit-$[TYPE].R") %>>%
+    "result-$[DATA]_$[TYPE].csv"
R> job <- expandTemplate(tmpl, variants)
```

The resulting `job` contains six rules that combine the specified variants as follows:

```
R> print(job)

[[1]]
(fit-lm.R, dataSimple.csv) -> R -> (result-dataSimple_lm.csv)
[[2]]
(fit-lm.R, dataComplex.csv) -> R -> (result-dataComplex_lm.csv)
[[3]]
(fit-rf.R, dataSimple.csv) -> R -> (result-dataSimple_rf.csv)
[[4]]
(fit-rf.R, dataComplex.csv) -> R -> (result-dataComplex_rf.csv)
[[5]]
(fit-nnet.R, dataSimple.csv) -> R -> (result-dataSimple_nnet.csv)
[[6]]
(fit-nnet.R, dataComplex.csv) -> R -> (result-dataComplex_nnet.csv)
```

If duplicated rules are created during the template expansion, they are omitted, as in the following job:

```
R> tmpl <- "data.csv" %>>%
+    rRule("pre.R") %>>%  "pre.rds" %>>%
+    rRule("comp.R", params=list(alpha="$[NUM]")) %>>% "result-$[NUM].csv"
R> variants <- data.frame(NUM=1:5)
R> job <- expandTemplate(tmpl, variants)
```

Expansion of the template would yield in repeating the rule

```
"data.csv" %>>% rRule("pre.R") %>>%  "pre.rds"
```

However, the repeated rules are automatically removed as can be seen from the print:

```
R> print(job)

[[1]]
(pre.R, data.csv) -> R -> (pre.rds)
[[2]]
(comp.R, pre.rds) -> R -> (result-1.csv)
[[3]]
(comp.R, pre.rds) -> R -> (result-2.csv)
[[4]]
(comp.R, pre.rds) -> R -> (result-3.csv)
[[5]]
(comp.R, pre.rds) -> R -> (result-4.csv)
[[6]]
(comp.R, pre.rds) -> R -> (result-5.csv)
```

On the other hand, (not only) a template expansion may often result in distinct rules producing a duplicated target. Such sequence of rules is prohibited and causes an error message in the `makefile()` function. The problem is illustrated in the example below.

```
R> tmpl <- "data-$[TYPE].csv" %>>% markdownRule("report.Rmd") %>>%
+      "report.pdf"
R> variants <- data.frame(TYPE=c("a", "b", "c"))
R> job <- expandTemplate(tmpl, variants)
R> print(job)

[[1]]
(report.Rmd, data-a.csv) -> markdown -> (report.pdf)
[[2]]
(report.Rmd, data-b.csv) -> markdown -> (report.pdf)
[[3]]
(report.Rmd, data-c.csv) -> markdown -> (report.pdf)
```

Here the three different rules produce the same target (`report.pdf`). An attempt to generate the `Makefile` would end with an error:

```
R> makefile(job)

Error in .validate(job):  Duplicate targets found:  report.pdf
```

6. Conclusion

The presented **rmake** package provides an easy but powerful way for managing complex data manipulation processes in R using the well-known and broadly adopted *Make* utility. **rmake** brings tools for generation of the `Makefile`, in which the file dependencies and build rules are defined. Advanced features of **rmake** such as pipelining (`%>>%`), parameterized rules, or rule templates, enable quick definition of that file dependencies.

Acknowledgements

The support by the project “LQ1602 IT4Innovations excellence in science” is gratefully acknowledged.

References

Allaire J, Xie Y, McPherson J, Luraschi J, Ushey K, Atkins A, Wickham H, Cheng J, Chang W (2017). *rmarkdown: Dynamic Documents for R.* R package version 1.8, URL `https://CRAN.R-project.org/package=rmarkdown`.

Landau WM (2018). *drake: A Pipeline Toolkit for Reproducible Computation at Scale.* R package version 6.0.0, URL `https://CRAN.R-project.org/package=drake`.

R Core Team (2017). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL `https://www.R-project.org/`.

RStudio Team (2015). *RStudio: Integrated Development Environment for R.* RStudio, Inc., Boston, MA. URL `http://www.rstudio.com/`.

Xie Y (2015). *Dynamic Documents with R and knitr.* 2nd edition. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1498716963, URL `https://yihui.name/knitr/`.

Xie Y (2017). *knitr: A General-Purpose Package for Dynamic Report Generation in R.* R package version 1.18, URL `https://yihui.name/knitr/`.

Affiliation:

Michal Burda
Institute for Research and Applications of Fuzzy Modeling
Centre of Excellence IT4Innovations, Division University of Ostrava
30. dubna 22, 701 03 Ostrava, Czech Republic
E-mail: `Michal.Burda@osu.cz`