

{targets}

# What is `{targets}`?



“a Make-like pipeline tool for statistics and data science in R”

- manage a sequence of computational steps
- only update what needs updating
- ensure that the results at the end of the pipeline are still valid

# Script-based workflow

## 01-data.R

```
1 library(tidyverse)
2 data <- read_csv("data.csv", col_types = cols()) %>%
3   filter(!is.na(Ozone))
4 write_rds(data, "data.rds")
```

## 02-model.R

```
1 library(tidyverse)
2 data <- read_rds("data.rds")
3 model <- lm(Ozone ~ Temp, data) %>%
4   coefficients()
5 write_rds(model, "model.rds")
```

## 03-plot.R

```
1 library(tidyverse)
2 model <- read_rds("model.rds")
3 data <- read_rds("data.rds")
4 ggplot(data) +
5   geom_point(aes(x = Temp, y = Ozone)) +
6   geom_abline(intercept = model[1], slope = model[2])
7 ggsave("plot.png", plot)
```

# Problems with script-based workflow

- **Reproducibility:** if you change something in one script, you have to remember to re-run the scripts that depend on it
- **Efficiency:** that means you'll usually rerun all the scripts even if they don't depend on the change
- **Scalability:** if you have a lot of scripts, it's hard to keep track of which ones depend on which
- **File management:** you have to keep track of which files are inputs and which are outputs and where they're saved

# # {targets}: The basics

# {targets} workflow

## R/functions.R

```
1 get_data <- function(file) {  
2   read_csv(file, col_types = cols()) %>%  
3   filter(!is.na(Ozone))  
4 }  
5  
6 fit_model <- function(data) {  
7   lm(Ozone ~ Temp, data) %>%  
8   coefficients()  
9 }  
10  
11 plot_model <- function(model, data) {  
12   ggplot(data) +  
13     geom_point(aes(x = Temp, y = Ozone)) +  
14     geom_abline(intercept = model[1,1], slope = model[2,1])
```

# {targets} workflow

\_targets.R

```
1 library(targets)
2
3 tar_source()
4 tar_option_set(packages = c("tidyverse"))
5
6 list(
7   tar_target(file, "data.csv", format = "file"),
8   tar_target(data, get_data(file)),
9   tar_target(model, fit_model(data)),
10  tar_target(plot, plot_model(model, data))
11 )
```

Run `tar_make()` to run pipeline



`use_targets()` will generate a `_targets.R` script for you to fill in.



# {targets} workflow

Targets are “hidden” away where you don’t need to manage them

```
|— _targets.R
|— data.csv
|— R/
|   |— functions.R
|— _targets/
|   |— objects
|       |— data
|       |— model
|       |— plot
```



You can of course have multiple files in `R/`; `tar_source()` will source them all

# My typical workflow with `{targets}`

1. Read in some data and do some cleaning until it's in the form I want to work with.
2. Wrap that in a function and save the file in `R/`.
3. Run `use_targets()` and edit `_targets.R` accordingly, so that I list the data file as a target and `clean_data` as the output of the cleaning function.
4. Run `tar_make()`.
5. Run `tar_load(clean_data)` so that I can work on the next step of my workflow.
6. Add the next function and corresponding target when I've solidified that step.



I usually include `library(targets)` in my project `.Rprofile` so that I can always call `tar_load()` on the fly

# \_targets.R tips and tricks

```
1 list(  
2   tar_target(  
3     data_file,  
4     "data/raw_data.csv",  
5     format = "file"  
6   ),  
7   tar_target(  
8     raw_data,  
9     read.csv(data_file)  
10  ),  
11  tar_target(  
12    clean_data,  
13    clean_data_function(raw_data)  
14  ),  
15  )
```



# \_targets.R tips and tricks

```
1 preparation <- list(  
2   ...,  
3   tar_target(  
4     clean_data,  
5     clean_data_function(raw_data)  
6   )  
7 )  
8 modeling <- list(  
9   tar_target(  
10    linear_model,  
11    linear_model_function(clean_data)  
12  ),  
13  ...  
14 )
```



By grouping the targets into lists, I can easily comment out chunks of the pipeline to not run the whole thing

# \_targets.R tips and tricks

```
1 ## prepare ----
2 prepare <- list(
3   ### cleanData.csv ----
4   tar_target(
5     cleanData.csv,
6     file.path(path_to_data,
7               "cleanData.csv"),
8     format = "file"
9   ),
10  ### newdat ----
11  tar_target(
12    newdat,
13    read_csv(cleanData.csv,
14             guess_max = 20000)
```

prepare	raw
cleanData.csv	log_res_raw
newdat	mult_res_raw
fulldat	ctc_res_raw
flow	tte_res_raw
loglinear	tte_res_raw_very
log_dat	all_res_raw
log_dat_covid_mild...	imputation
log_dat_covid	mice_params
log_est	cols_to_impute
log_est_severity	fulldat_imputed
loglinear_spon	log_dat_imputed
log_est_spon	mice_comparison
log_est_severity_spon	tabs_figs
log_est_ind	descriptive_table
log_est_severity_ind	descriptive_table_o...
mult_est	outcomes_table
mult_est_severity	risk_plot
ctc	tte_plot
ctc_dat	followup_tab
cases	testing_plot
controls	all targets



In big projects, I comment my `_targets.R` file so that I can use the RStudio outline pane to navigate the pipeline (`my buggy function`)

# Key `{targets}` functions

- `use_targets()` gets you started with a `_targets.R` script to fill in
- `tar_make()` runs the pipeline and saves the results in `_targets/objects/`
- `tar_make_future()` runs the pipeline in parallel<sup>1</sup>
- `tar_load()` loads the results of a target into the global environment (e.g., `tar_load(clean_data)`)
- `tar_read()` reads the results of a target into the global environment (e.g., `dat <- tar_read(clean_data)`)
- `tar_visnetwork()` creates a network diagram of the pipeline
- `tar_outdated()` checks which targets need to be updated
- `tar_prune()` deletes targets that are no longer in `_targets.R`
- `tar_destroy()` deletes the `.targets/` directory if you need to burn everything down and start again

<sup>1</sup> Note: `{targets}` is moving to a new distributed computing strategy using `foreach`

Advanced {targets}



# “target factories”



repo status **Active** tarchetypes makes it easy to add certain kinds of common tasks to reproducible pipelines. Most of its functions create families of targets for parameterized R Markdown, simulation studies, and other general-purpose scenarios.



repo status **Active** stantargets is a workflow framework for Bayesian data analysis with cmdstanr. With concise, easy-to-use syntax, it defines versatile families of targets tailored to Bayesian statistics, from a single MCMC run with postprocessing to large simulation studies.



repo status **Active** Like stantargets, jagstargets is a workflow framework for Bayesian data analysis, with support for both single MCMC runs and large-scale simulation studies. It invokes JAGS through the R2jags package, which has nice features such as the ability to parallelize chains across local R processes.

# `{tarchetypes}`: reports

Render documents that depend on targets loaded with `tar_load()` or `tar_read()`.

- `tar_render()` renders an R Markdown document
- `tar_quarto()` renders a Quarto document (or project)



It can't detect dependencies like `tar_load(ends_with("plot"))`

# What does `report.qmd` look like?

```
1 ---
2 title: "My report"
3 ---
4 ```{r}
5 library(targets)
6 tar_load(results)
7 tar_load(plots)
8 ```
9 There were `r results$n` observations with a mean age of `r result
10 ```{r}
11 library(ggplot2)
12 plots$age_plot
13 ```
```

Because `report.qmd` depends on `results` and `plots`, it will only be re-rendered if either of those targets change.



The `extra_files =` argument can be used to force it to depend on additional non-target files

# {tarchetypes}: branching

Using data from the National Longitudinal Survey of Youth,

`_targets.R`

```
1 library(targets)
2 library(tarchetypes)
3 tar_source()
4
5 targets_setup <- list(
6   tar_target(
7     csv,
8     "data/nlsy.csv",
9     format = "file"
10  ),
11  tar_target(
12    dat,
13    readr::read_csv(csv,
14      show_col_types = FALSE)
15  )
16 )
```

`R/functions.R`

```
1 model_function <- function(outcome_var,
2                             sex_val, dat) {
3
4   lm(as.formula(paste(outcome_var,
5     " ~ age_bir + income + factor(region)")),
6     data = dat,
7     subset = sex == sex_val)
8 }
9
10 coef_function <- function(model) {
11   coef(model)[["age_bir"]]
12 }
```

we want to investigate the relationship between age at first birth and hours of sleep on weekdays and weekends among moms and dads separately

# Option 1

Create (and name) a separate target for each combination of sleep variable ("sleep\_wkdy", "sleep\_wknd") and sex (male: 1, female: 2):

```
1 targets_1 <- list(  
2   tar_target(  
3     model_1,  
4     model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat =  
5   ),  
6   tar_target(  
7     coef_1,  
8     coef_function(model_1)  
9   )  
10 )
```

... and so on...

# Option 2

Use `tarchetypes::tar_map()` to map over the combinations for you (static branching):

```
1 targets_2 <- tar_map(  
2   values = tidyr::crossing(  
3     outcome = c("sleep_wkdy", "sleep_wknd"),  
4     sex = 1:2  
5   ),  
6   tar_target(  
7     model_2,  
8     model_function(outcome_var = outcome, sex_val = sex, dat = dat  
9   ),  
10  tar_target(  
11    coef_2,  
12    coef_function(model_2)  
13  )  
14 )
```

## Option 2, cont.

Use `tarchetypes::tar_combine()` to combine the results of a call to `tar_map()`:

```
1 combined <- tar_combine(  
2   combined_coefs_2,  
3   targets_2[["coef_2"]],  
4   command = vctrs::vec_c(!!!.x),  
5 )  
6 tar_read(combined_coefs_2)
```

`command = vctrs::vec_c(!!!.x)` is the default, but you can supply your own function to combine the results



# Option 3

Use the `pattern =` argument of `tar_target()` (dynamic branching):

```
1 targets_3 <- list(  
2   tar_target(  
3     outcome_target,  
4     c("sleep_wkdy", "sleep_wknd")  
5   ),  
6   tar_target(  
7     sex_target,  
8     1:2  
9   ),  
10  tar_target(  
11    model_3,  
12    model_function(outcome_var = outcome_target, sex_val = sex_target,  
13    pattern = cross(outcome_target, sex_target)  
14  ),  
15)
```

# Branching

## Dynamic

---

Pipeline creates new targets at runtime.

---

Cryptic target names.

---

Scales to hundreds of branches.

---

No metaprogramming required.

## Static

---

All targets defined in advance.

---

Friendly target names.

---

Does not scale as easily for `tar_visnetwork()` etc.

---

Familiarity with metaprogramming is helpful.

# Branching

- The book also has an example of using metaprogramming to map over different functions
  - i.e. fit multiple models with the same arguments
- Static and dynamic branching can be combined
  - e.g. `tar_map(values = ..., tar_target(..., pattern = map(...)))`
- Branching can lead to slowdowns in the pipeline (see book for suggestions)

# {tarchetypes}: repetition

`tar_rep()` repeats a target multiple times with the same arguments

```
1 targets_4 <- list(  
2   tar_rep(  
3     bootstrap_coefs,  
4     dat |>  
5       dplyr::slice_sample(prop = 1, replace = TRUE) |>  
6       model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat  
7       coef_function()),  
8     batches = 10,  
9     reps = 10  
10  )  
11 )
```

The pipeline gets split into `batches` x `reps` chunks, each with its own random seed

# {tarchetypes}: mapping over iterations

```
1 sensitivity_scenarios <- tibble::tibble(  
2   error = c("small", "medium", "large"),  
3   mean = c(1, 2, 3),  
4   sd = c(0.5, 0.75, 1)  
5 )
```

`tar_map_rep()` repeats a target multiple times with different arguments

```
1 targets_5 <- tar_map_rep(  
2   sensitivity_analysis,  
3   dat |>  
4     dplyr::mutate(sleep_wkdy = sleep_wkdy + rnorm(nrow(dat), mean,  
5     model_function(outcome_var = "sleep_wkdy", sex_val = 1, dat =  
6     coef_function() |>  
7     data.frame(coef = _),  
8   values = sensitivity_scenarios,
```

```
9     batches = 10,  
10     reps = 10  
11 )
```

# {tarchetypes}: mapping over iterations

```
1 tar_read(sensitivity_analysis) |> head()
```

Ideal for sensitivity analyses that require multiple iterations of the same pipeline with different parameters

```
1 tar_read(sensitivity_analysis) |>
2   dplyr::group_by(error) |>
3   dplyr::summarize(q25 = quantile(coef, .25),
4                   median = median(coef),
5                   q75 = quantile(coef, .75))
```

# Conclusion

- `{targets}` is a great tool for managing complex workflows
- `{tarchetypes}` makes it even more powerful
- The [user manual](#) is a great resource for learning more



