
Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT
Data Science Pipelines with {targets}
[Dr. Hanjo Odendaal]
Photo by Stefan Els

Hanjo Odendaal

PRINCIPAL DATA SCIENTIST (71POINT4)

ABOUT ME

With over 10 years experience working throughout Africa, Hanjo leads 71point4's analytics, statistics and engineer aspects on projects. His expertise focuses on how data can be employed to answer impactful and meaningful questions that always aim to put people at the center.

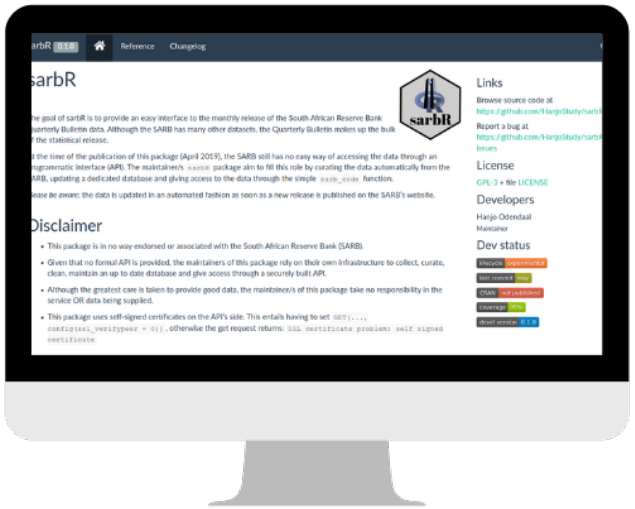
He has worked in banking, insurance, housing, agriculture, telecommunications and credit; focusing mostly on desigining and managing the implementation of analytical solutions to deliver real-time insights for the client.

Hanjo holds a PhD in Economics from the University of Stellenbosch. His thesis focused on the application of natural language processing and statistical learning applied within traditional economic frameworks.

Hanjo Odendaal

PRINCIPAL DATA SCIENTIST (71POINT4)

Software Engineering



Production Machine Learning

Practical Machine Learning

H_2O.ai

High Performance Cloud Computing

R + amazon web services™

Web Scraping

useR!

Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Defence against the dark arts a.k.a code entropy

Why pipelines?

> When writing code, there are always two people involved: You and Future You... and about 90% of the time Future You is going to hate how you structured your project 6 months ago.

Without a build automation tool, a pipeline is nothing but a series of scripts that get called one after the other. In most cases when students start out, the pipeline tends to be one very long script that is usually a 🤞 and hope as scaffolding that ensures operations successfully.

- Scripts can, and will, be executed out of order.
  - if you make a small change in one of your scripts, a logic execution needs to follow to ensure everything is executed in the correct order.
- Pipelines written as scripts are usually quite difficult to read and understand.
  - Comments help, but has to also be updated as you update your scripts.

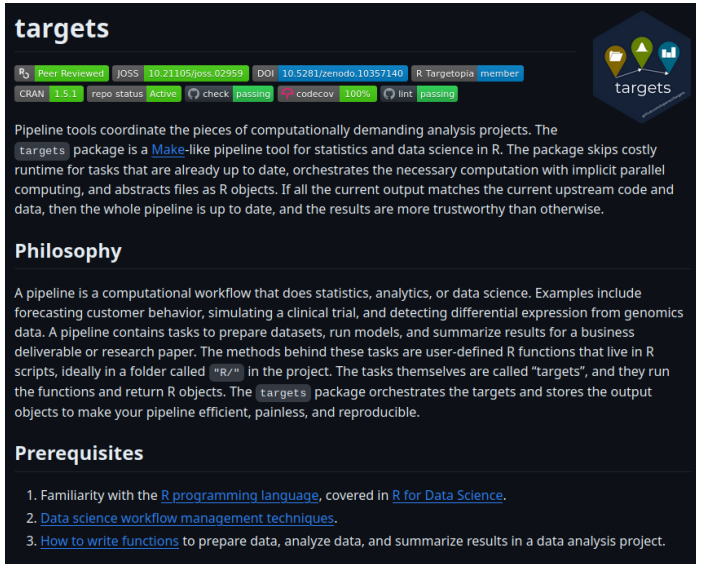
These are just some of the basic issues... BUT luckily you have heard of the term "functional" programming. This part of the course will show you how to supercharge this concept 🚀!

Learning outcomes

> Now and then, the bright and shiny objects that we stumble across can turn out to be very useful.

We want to take you on a high-level tour of `{targets}`. Getting you started with the basics! The rest of the journey is up to you:

- What is `{targets}`?
- Integrating databases into your {targets} pipeline
- NLP and `{targets}`



targets

Peer Reviewed | JOSS 10.21105/joss.02959 | DOI 10.5281/zenodo.10357140 | R Targetopia member
CRAN 1.5.1 | repo status Active | check passing | codecov 100% | lint passing

Pipeline tools coordinate the pieces of computationally demanding analysis projects. The `targets` package is a Make-like pipeline tool for statistics and data science in R. The package skips costly runtime for tasks that are already up to date, orchestrates the necessary computation with implicit parallel computing, and abstracts files as R objects. If all the current output matches the current upstream code and data, then the whole pipeline is up to date, and the results are more trustworthy than otherwise.

Philosophy

A pipeline is a computational workflow that does statistics, analytics, or data science. Examples include forecasting customer behavior, simulating a clinical trial, and detecting differential expression from genomics data. A pipeline contains tasks to prepare datasets, run models, and summarize results for a business deliverable or research paper. The methods behind these tasks are user-defined R functions that live in R scripts, ideally in a folder called `"R/"` in the project. The tasks themselves are called "targets", and they run the functions and return R objects. The `targets` package orchestrates the targets and stores the output objects to make your pipeline efficient, painless, and reproducible.

Prerequisites

1. Familiarity with the R programming language, covered in R for Data Science.
2. Data science workflow management techniques.
3. How to write functions to prepare data, analyze data, and summarize results in a data analysis project.

What is {targets}?

It is a workflow management package for the R programming language developed and maintained by Will Landau. If some of you program `python` / `bash` you might be familiar or have heard about `snakemake` or `make` files. `{targets}` is the `R` equivalent of `make` files as you will see when we run `tar_make()` to run our pipeline.

The major features of targets include:

- Automation of workflow
- Caching of workflow steps
- Batch creation of workflow steps
- Parallelization at the level of the workflow

Why is this amazing?

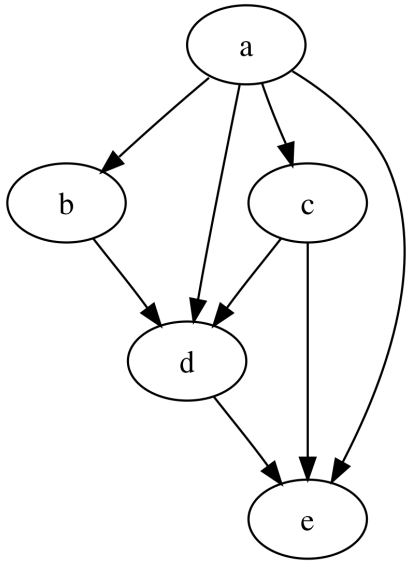
- Return to a project after working on something else and immediately pick up where you left off without confusion or trying to remember what you were doing
- If the workflow changes in any way, only re-run the parts that that are affected by the change
- Massively scale up the workflow without changing individual functions through dynamic branching and parallelization (advanced)

All of these characteristics of pipelines give us a high probability of ⭐reproducibility⭐!

What is a DAG?

DAG or Directed acyclic graph, is a mathematical concept. This "DAG" consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

You are constantly thinking about 3 things: *Input* > *function* > *output*.



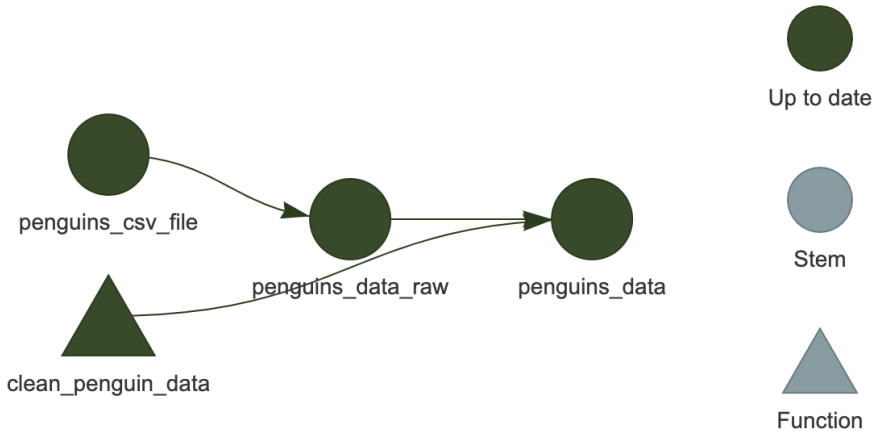
What is a DAG?

DAG or Directed acyclic graph, is a mathematical concept. This "DAG" consists of vertices and edges (also called arcs), with each edge directed from one vertex to another, such that following those directions will never form a closed loop.

penguin_csv_file `>`

penguins_data_raw(penguin_csv_file) `>`

penguins_data(penguins_data_raw)



Ready for the basics!

Time for the setup

Please take a couple of minutes to install these packages (`Ncpu` parameter for parallel installation):

Pipeline

```
install.packages("igraph", Ncpus = 4)
install.packages("visNetwork", Ncpus = 4)
install.packages("targets", Ncpus = 4)
install.packages("tarchetypes", Ncpus = 4)
```

Utilies

```
install.packages("tidyverse", Ncpus = 4)
install.packages("glue", Ncpus = 4)
install.packages("logger", Ncpus = 4)
install.packages("lubridate", Ncpus = 4)
```

Modeling

```
install.packages("ranger", Ncpus = 4)
```

NLP

```
install.packages("quanteda", Ncpus = 4)
install.packages("topicmodels", Ncpus = 4)
```

10:00

Time for the setup

⚠️ You always have to work in a `Project` when working with `{targets}` otherwise its not going to work! We will use the `Texevier` to help us setup that folder.

```
library(Texevier)
# Input your directory as dir:
dir ← "~/Downloads/"

template_name = "targets_example"
create_template(directory = dir,
                template_name = template_name,
                build_project = TRUE,
                open_project = FALSE)
```

```
(base) hanjo@optimus:~/Downloads/targets_example$ tree
.
├── code
│   ├── Data_Create.R
│   └── Example_Plot_Scatter.R
├── data
├── targets_example.Rmd
├── targets_example.Rproj
└── Tex
    ├── EconEJ.bst
    ├── elsevier-harvard.csl
    ├── harvard1.csl
    ├── harvard-stellenbosch-university.csl
    ├── packages.txt
    ├── ref.bib
    └── TexDefault.txt
```

Sidequest 💎💰👑🎁

If you want your prompt to show you what Github branch you are on, add the following to your `.Rprofile`. Easiest is `usethis::edit_r_profile()`:

```
if(requireNamespace("prompt", quietly = TRUE)){
# devtools::install_github("gaborcsardi/prompt@x")
 prompt_git ← function( ... ){
    paste0(
      "[", prompt::git_branch(), "]", ">"
    )
  }
  prompt::set_prompt(prompt_git)
  rm(prompt_git)
}
```

Time for the setup

After you have ensured that you can `knit` the file called: `targets_example.Rmd`, create a file called `_targets.R` in the parent folder. Take special note of the `_` in the filename. Also create a `README.qmd`!

```r
#_targets.R

# 1.0 Setup --------------------------------
options(tidyverse.quiet = TRUE)

library(targets)
library(tarchetypes)

# * Libraries ------------------------------
tar_option_set(packages = c(
  # data
  "tidyverse",
  "lubridate",
  "glue",
  "logger",
  # modeling
  "plm",
  # reporting
  "huxtable"
))

# Source all of your functions into environment
sapply(list.files("code", full.names = T), source)
```

````
# README.qmd

---
title: "My readme"
format: html
editor_options:
  chunk_output_type: console
execute:
  echo: false
  eval: false
---

```{r}
library(tidyverse)
library(targets)
```

Execute the pipeline using `tar_make`:

```{r}
tar_make()
```
```
````
````

Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Building first pipeline ⚙️

Data

The most obvious step in our data analysis is obviously getting our data into R.

Golden rule:

- If it can be avoided, never transform your data in Excel
- Use R to showcase how you extracted, loaded and transformed (ELT) your data

In your `_targets.R` file, add the following "targets":

```
# 2.0 Data ---------------------------------
target_data_load ← list(
  tar_target(input_file,
             "data/Example_data.rds"),
  tar_target(examples_df,
             read_rds(input_file))
)

c(target_data_load)
```

Data target breakdown

I assign an object of `list()` class to my first *grouping* of targets.

```
target_data_load ← list()
```

In my first target I say that the input of "data/Example_data.rds" should be assigned to `input_file`:

```
tar_target(input_file,"data/Example_data.rds")
```

Secondly I then use `input_file` as my `input` in the function `read_rds` and call that output, `example_df`:

```
tar_target(examples_df, read_rds(input_file)
```

Lastly I concatenate all my groups together:

```
c(target_data_load)
```

^ *I always use a suffix to help me know what the output type is:* `*_df`, `*_list`, `*_plot`, `*_write`

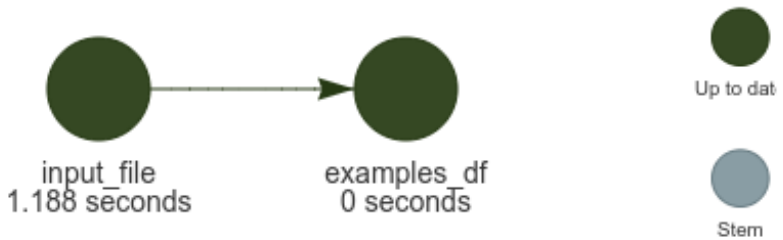
Data

Once you have completed the input into your `_targets.R` file, go to your `Readme.qmd` and run the `tar_make()` command:

```
[main]>tar_make()
▶ start target input_file
● built target input_file [1.188 seconds]
▶ start target examples_df
● built target examples_df [0 seconds]
▶ end pipeline [1.249 seconds]
```

Ok, so what happened and what now? Well, congrats on your running your first pipeline!! 🎉

To see your pipeline, add the command `tar_visnetwork(targets_only = TRUE, label = "time")` to your `Readme.qmd`... you should see the your DAG once you have run it.



Accessing the objects

Loading data into the pipeline is all good, but how can we use it? There is primarily two commands: `tar_read` and `tar_load`.

In the *first* instance you can imagine its like reading in a `csv`:

```
[main]>out ← tar_read(examples_df)
[main]>print(out)
# A tibble: 100 × 3
   Height_Score Weight_Score Agility_Score
          <dbl>        <dbl>         <dbl>
 1         5.75         6.11             3
 2         3.84         3.37             3
 3         5.71         2.86             3
 4         6.02         5.59             3
 5         6.56         6.48             3
```

Where in the case of `tar_load` the object is loaded into your environment and you can then use it:

```
[main]>tar_load(examples_df)
[main]>print(examples_df)
```

How much a target should do?

The `{targets}` package automatically skips targets that are already up to date, so it is best to define targets that maximize time savings and breaks down the analytical work into succinct problem definitions: (1) Read, (2) Augement, (3) Analyse etc. Good targets usually:

- Invoke no side effects such as modifications to the global environment!
- Are large enough to subtract a decent amount of runtime when skipped
- Are small enough that some targets can be skipped even if others need to run
- Return a single value that is:
  - Easy to understand and introspect.
  - Meaningful to the project.
  - Easy to save as a file

Think of organising your pipeline as a package. 📦

Adding modeling component to pipeline

I would bet most pipelines dont just want to read data, but analyse them. So, how do we do that? Well lets start by creating a new script for our "modeling" functions: `code/models.R`:

Basic Model:

```
model_lm ← function(examples_df){
    lm(Agility_Score ~., data = examples_df)
}
```

Interaction Model:

```
model_lm_interactions ← function(examples_df){
    lm(Agility_Score ~ Height_Score*Weight_Score, data = examples_df)
}
```

[*] *Take note of how I specify the input name the same as my previous output. You will thank me later ;-)*

Adding modeling component to pipeline

Once you have saved the R file, we need to now add the functions to our pipeline in `_targets.R`:

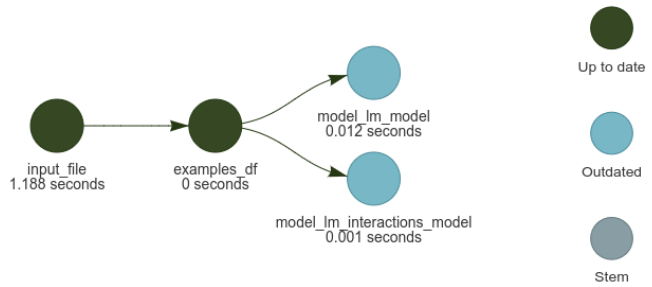
```
# 2.0 Data --------------------------------
target_data_load ← list(
    tar_target(input_file,
               "data/Example_data.rds"),
    tar_target(examples_df,
               read_rds(input_file))
)

# 3.0 Model -------------------------------
target_model ← list(
    tar_target(model_lm_model,
               model_lm(examples_df)),
    tar_target(model_lm_interactions_model,
               model_lm_interactions(examples_df))
)

c(target_data_load,
  target_model)
```

Adding modeling component to pipeline

Now that you have added in the new targets. Have a look at your DAG:



Then, you have guessed it, time to run `tar_make()`:

```
[main]>tar_make()
✔ skip target input_file
✔ skip target examples_df
▶ start target model_lm_model
● built target model_lm_model [0.012 seconds]
▶ start target model_lm_interactions_model
● built target model_lm_interactions_model [0.001 seconds]
▶ end pipeline [1.213 seconds]
```

Visualise results

Models in themselves arent that useful. Lets extract the RMSE using `yardstick` to compare which model was a better fit and then compare them visually. Because we want to apply the same action on the model, we only going to need a single function! In the `code/models.R` file lets add a function to extract the models' RMSE.

```
extract_rmse ← function(model, examples_df, model_name){

    out ← tibble(truth = examples_df$Agility_Score,
           prediction = predict(model, examples_df))
    model_summary ← yardstick::rmse(out, truth, prediction) %>%
        janitor::clean_names() %>%
        mutate(model_name = model_name)

    return(model_summary)
}
```

Visualise results

In `_targets.R`:

```
target_model ← list(
    tar_target(model_lm_model,
               model_lm(examples_df)),
    tar_target(model_lm_interactions_model,
               model_lm_interactions(examples_df)),
    tar_target(model_lm_model_rmse,
               extract_rmse(model_lm_model,
                            examples_df,
                            model_name = "model_lm")),
    tar_target(model_lm_interactions_model_rmse,
               extract_rmse(model_lm_interactions_model,
                            examples_df,
                            model_name = "model_lm_interactions"))

)
```

Visualise results

Now lets plot the RMSE!

```
plot_rmse ← function(model_lm_model_rmse,
                     model_lm_interactions_model_rmse){

    bind_rows(model_lm_interactions_model_rmse,
              model_lm_model_rmse) %>%
        mutate(model_name = snakecase::to_title_case(model_name)) %>%
        ggplot(., aes(model_name, estimate,
                      color = model_name, pch = model_name)) +
        geom_point(size = 5) +
        scale_color_grey(
            start = 0.2,
            end = 0.6
        ) +
        labs(
            y = "RMSE",
            x = "Model Specification",
            color = ""
        ) +
        ggthemes::theme_stata(base_size = 20) +
        theme(plot.background = element_rect(fill = "white")) +
        theme(legend.position = "bottom") +
        guides(pch = "none")
}
```

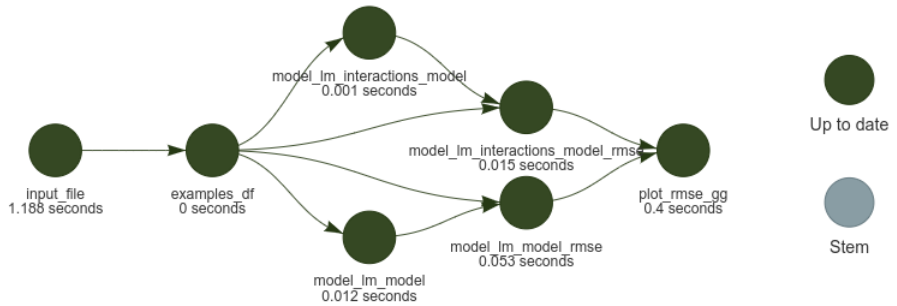
```
# 3.0 Model ------------------------------
target_model ← list(
    tar_target(model_lm_model,
               model_lm(examples_df)),
    tar_target(model_lm_interactions_model,
               model_lm_interactions(examples_df)),
    tar_target(model_lm_model_rmse,
               extract_rmse(model_lm_model,
                            examples_df,
                            model_name = "model_lm")),
    tar_target(model_lm_interactions_model_rmse,
               extract_rmse(model_lm_interactions_model,
                            examples_df,
                            model_name = "model_lm_interactions")),
    tar_target(plot_rmse_gg,
               plot_rmse(model_lm_model_rmse,
                         model_lm_interactions_model_rmse))
)
```

Visualise results

Now lets plot the RMSE!

```
plot_rmse ← function(model_lm_model_rmse,
                     model_lm_interactions_model_rmse){

    bind_rows(model_lm_interactions_model_rmse,
              model_lm_model_rmse) %>%
        mutate(model_name = snakecase::to_title_case(model_name)) %>%
        ggplot(., aes(model_name, estimate,
                      color = model_name, pch = model_name)) +
        geom_point(size = 5) +
        scale_color_grey(
            start = 0.2,
            end = 0.6
        ) +
        labs(
            y = "RMSE",
            x = "Model Specification",
            color = ""
        ) +
        ggthemes::theme_stata(base_size = 20) +
        theme(plot.background = element_rect(fill = "white")) +
        theme(legend.position = "bottom") +
        guides(pch = "none")
}
```

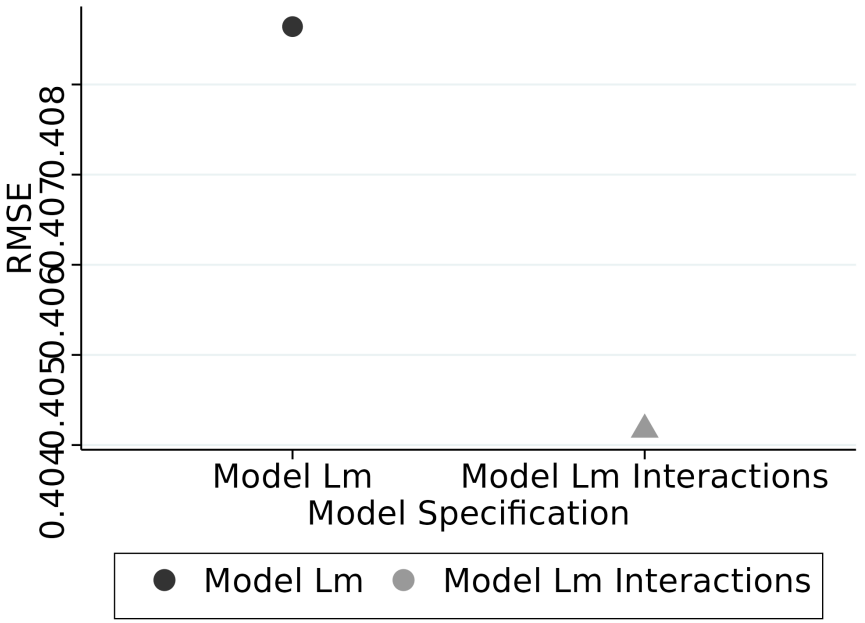


Visualise results

Now lets plot the RMSE!

```
plot_rmse ← function(model_lm_model_rmse,
                     model_lm_interactions_model_rmse){

    bind_rows(model_lm_interactions_model_rmse,
              model_lm_model_rmse) %>%
        mutate(model_name = snakecase::to_title_case(model_name)) %>%
        ggplot(., aes(model_name, estimate,
                      color = model_name, pch = model_name)) +
        geom_point(size = 5) +
        scale_color_grey(
            start = 0.2,
            end = 0.6
        ) +
        labs(
            y = "RMSE",
            x = "Model Specification",
            color = ""
        ) +
        ggthemes::theme_stata(base_size = 20) +
        theme(plot.background = element_rect(fill = "white")) +
        theme(legend.position = "bottom") +
        guides(pch = "none")
}
```



Integrate all into report

So why did we start with `Texevier`? Because we want our article to update if we make a change... the last piece of the puzzle is to render the `article.Rmd`:

Step 1: Make sure you have `library(targets)` in your setup chunk.

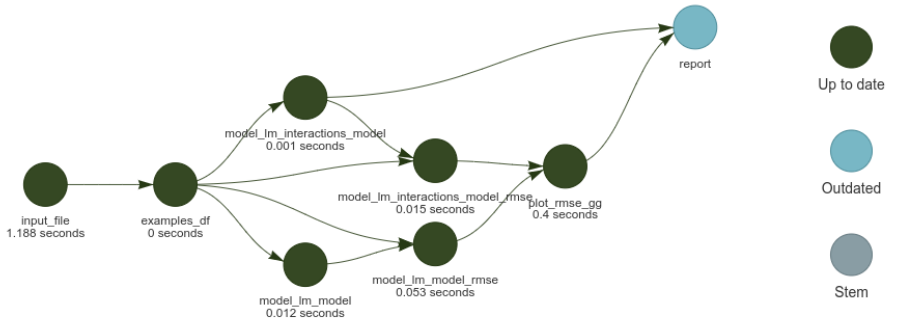
Step 2: Use `tar_read` or `tar_load` in `article.Rmd`.

Step 3: Build the render into your pipeline `_targets.R`:

```
# 4.0 Model ------------------------------
target_render ← list(
    tar_render(report, "article.Rmd")
)

c(target_data_load,
  target_model,
  target_render)
```

Step 4: View your network to see that the correct outputs go in article:



Step 5: Run `tar_make()`, sit back and marvel at your work in glory.

Are you not entertained?

If its good enough for Taylor* its good enough for you...

*approximation

Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Homework 🏡📝 (jip)

Homework

Using the techniques of today, transform the "script" based flow from the practical Tidy Regressions into a pipeline.

The assignment is simple:

- Create a new project
- Setup project for `{targets}`
- Change the "chunks" into functions
- Create a pipeline to bring in data, run models and output plot
- Run pipeline from section `Load Data` to `Backward Selection`

... in the next lesson we will be introducing the concept of "databases".