UQÀM | co-operators

Chaire Co-operators en
analyse des risques actuariels

Améliorer son flux de travail en `R` avec `targets`

Réunion mensuelle de la Chaire

Francis Duval
Université du Québec à Montréal
30 mars 2022

Qu'est-ce qu'un projet de science des données ?

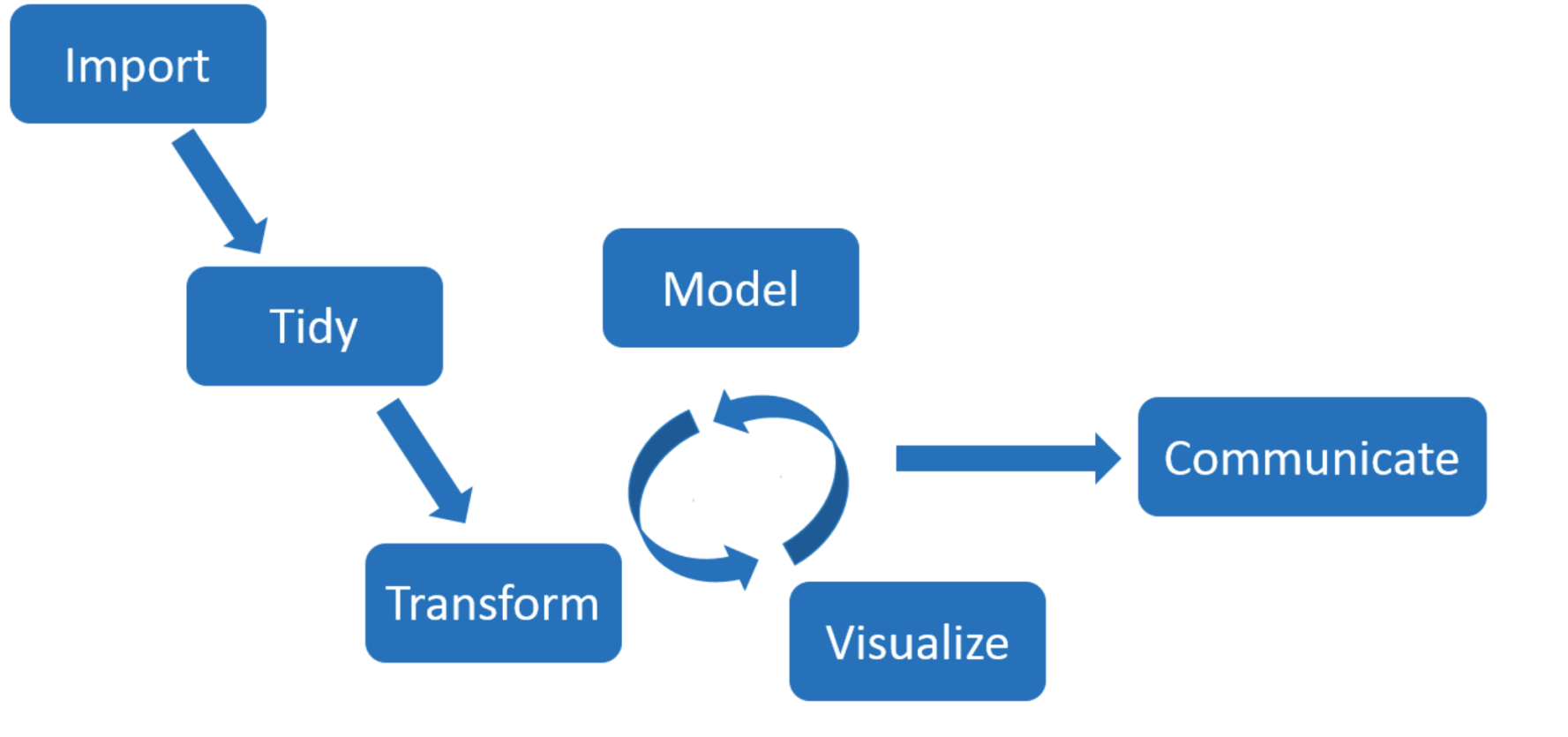
« Pipeline » typique d'un projet de science des données

1. Extraction des données
2. Préparation des données
3. Analyse descriptive et pré-modélisation
4. Modélisation, optimisation des hyperparamètres
5. Visualiser les résultats

Caractéristiques courantes d'un projet de science des données

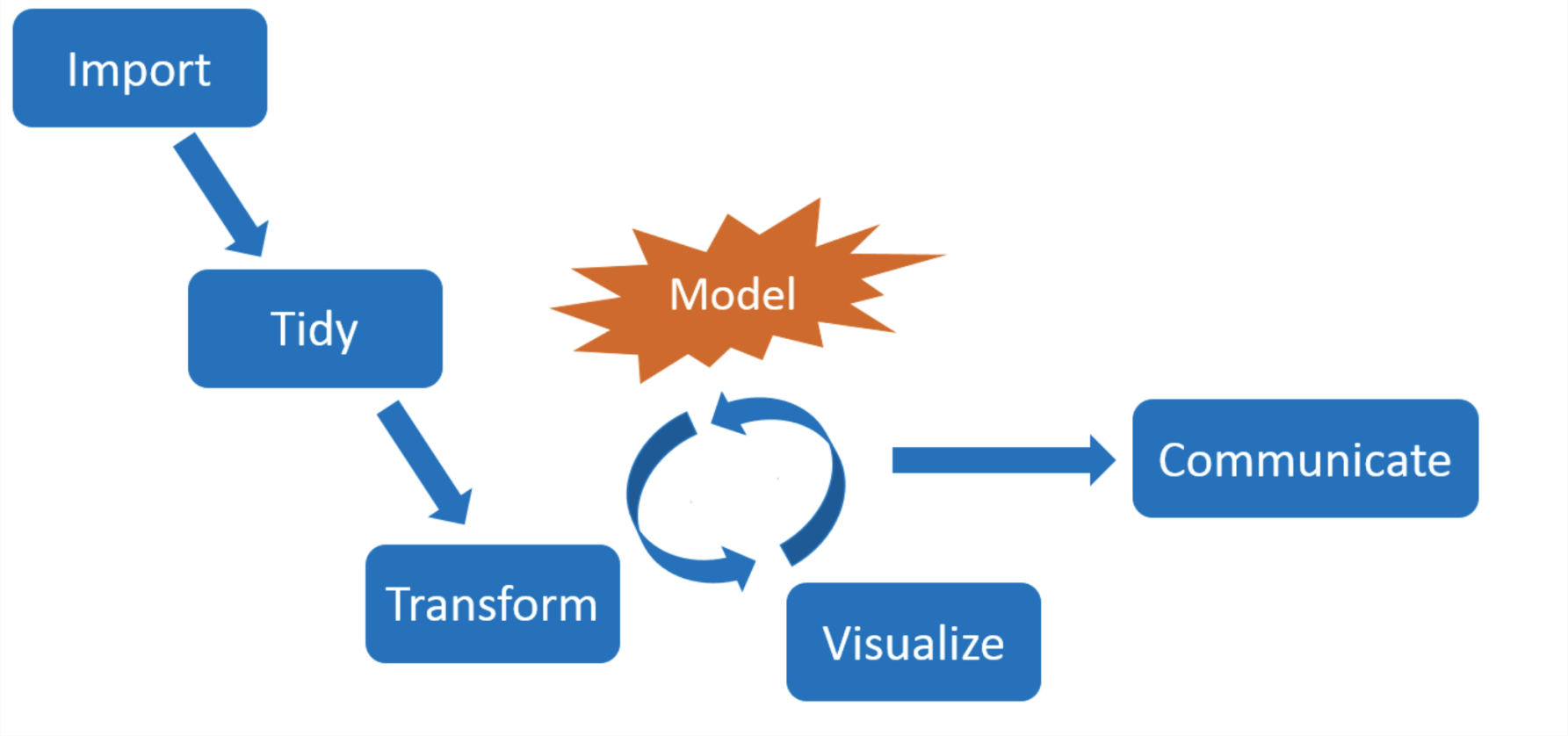
- Beaucoup de tâches interconnectées
- Certaines tâches demandent beaucoup de calcul
  - Machine learning
  - Réseaux de neurones
  - Bootstrap
  - Analyse bayésienne
  - Etc.
- Changements fréquents dans le code et dans les données

Tâches interconnectées



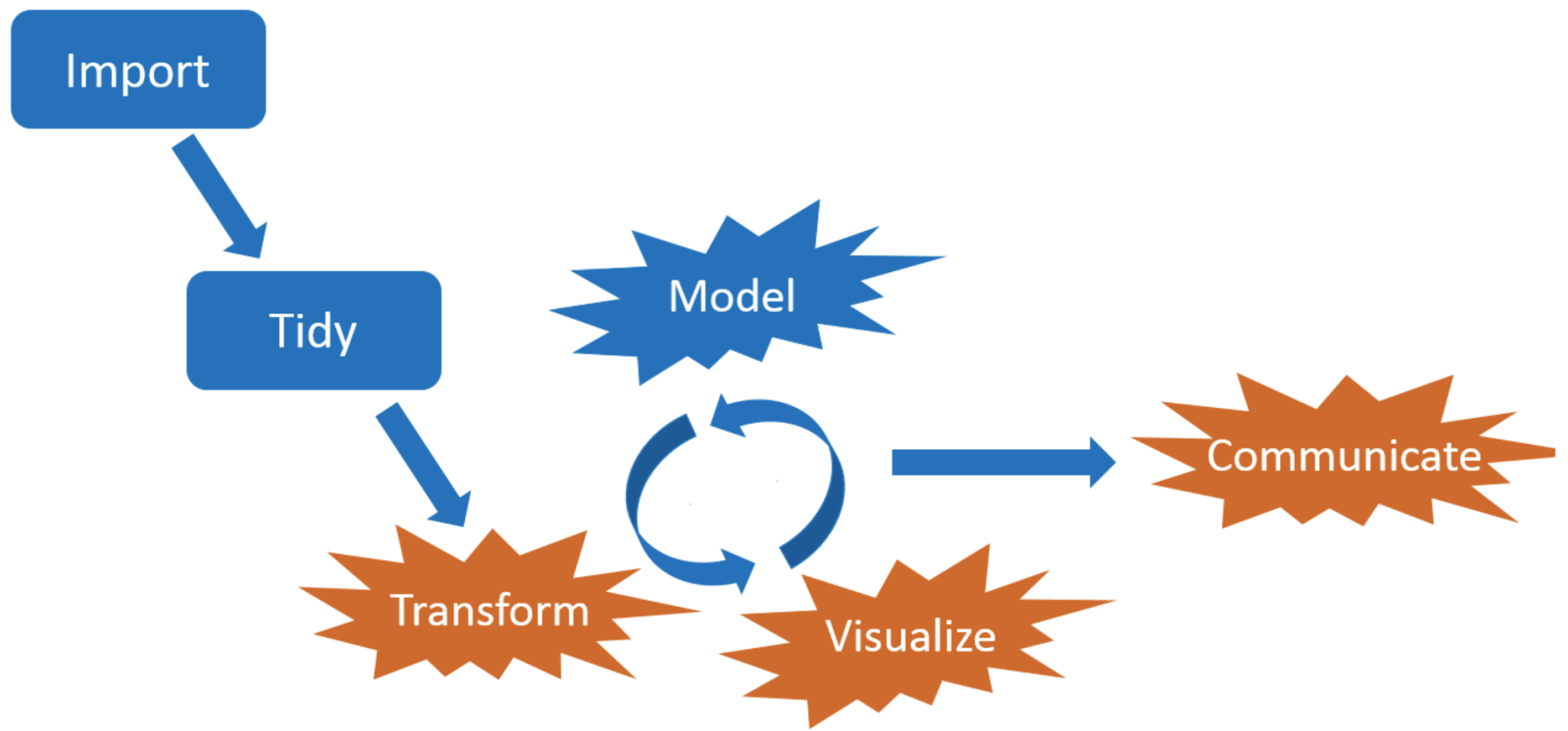
- Un projet comprend plusieurs tâches interconnectées

Changements



- Changement dans les modèles
- Correction d'un bug
- Changement dans les données
- Etc.

Conséquences



- Toutes les étapes en aval doivent être reroulées pour être mises à jour.

Qu'est-ce que `targets` ?

Quelques infos

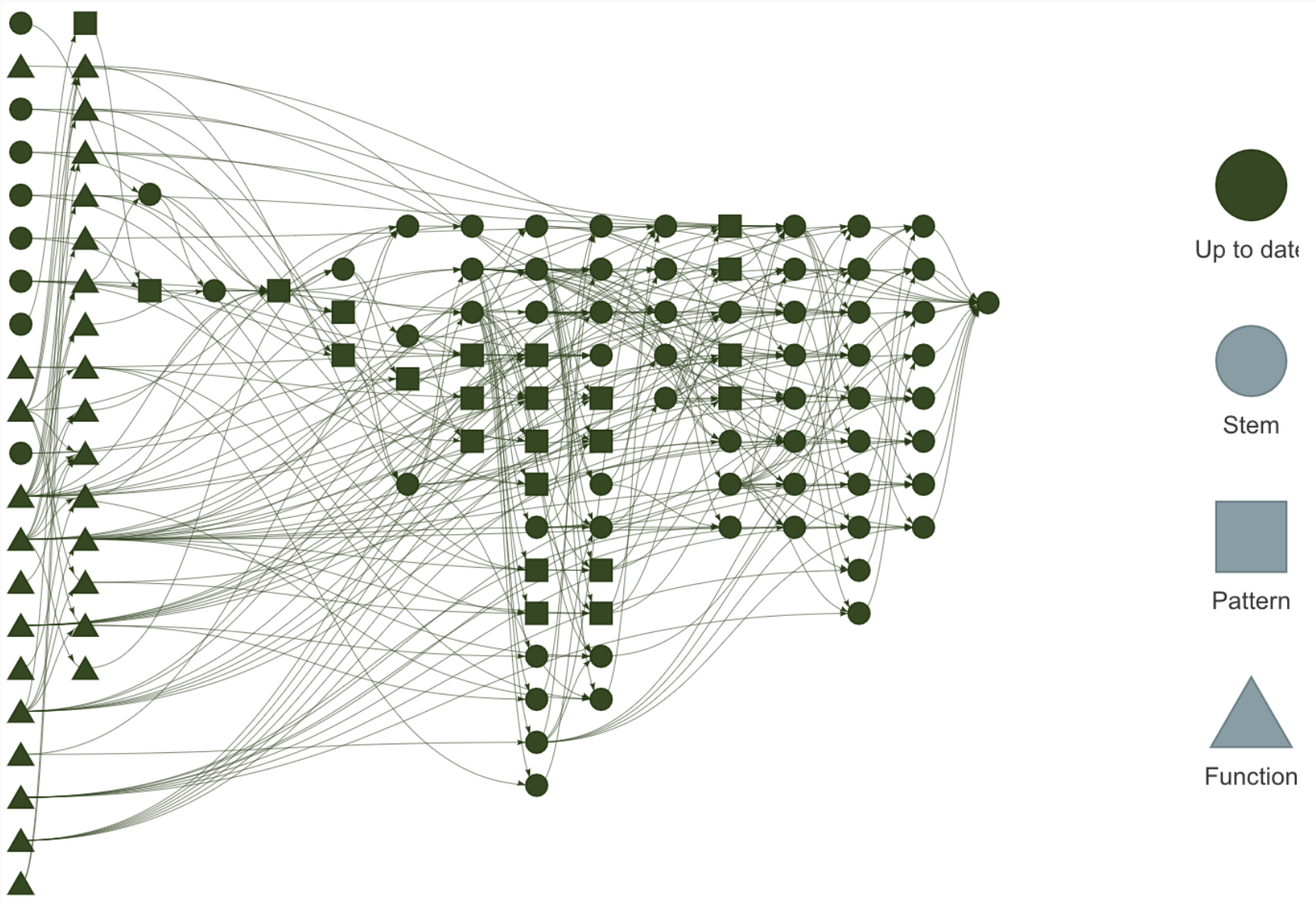
- Package `R` pour la **gestion de workflow** développé par Will Landau
- Successeur du package `drake`
- Sur le site du CRAN depuis un peu plus d'un an

Un projet de science des données peut prendre plusieurs heures (ou même jours, semaines) à rouler. On ne veut pas le rerouler depuis le début à chaque petite modification. Le package `targets` va créer un **graphe de dépendance** du projet. Lorsqu'un changement est fait, `targets` va seulement rouler les éléments du pipeline qui sont désynchronisés. C'est un outil incroyable pour

- nous forcer à écrire du **meilleur code**,
- réduire notre **charge mentale**,
- assurer la **reproductibilité** du workflow,
- augmenter la **fiabilité** des résultats,
- avoir une meilleure **vue d'ensemble** du pipeline (projet).

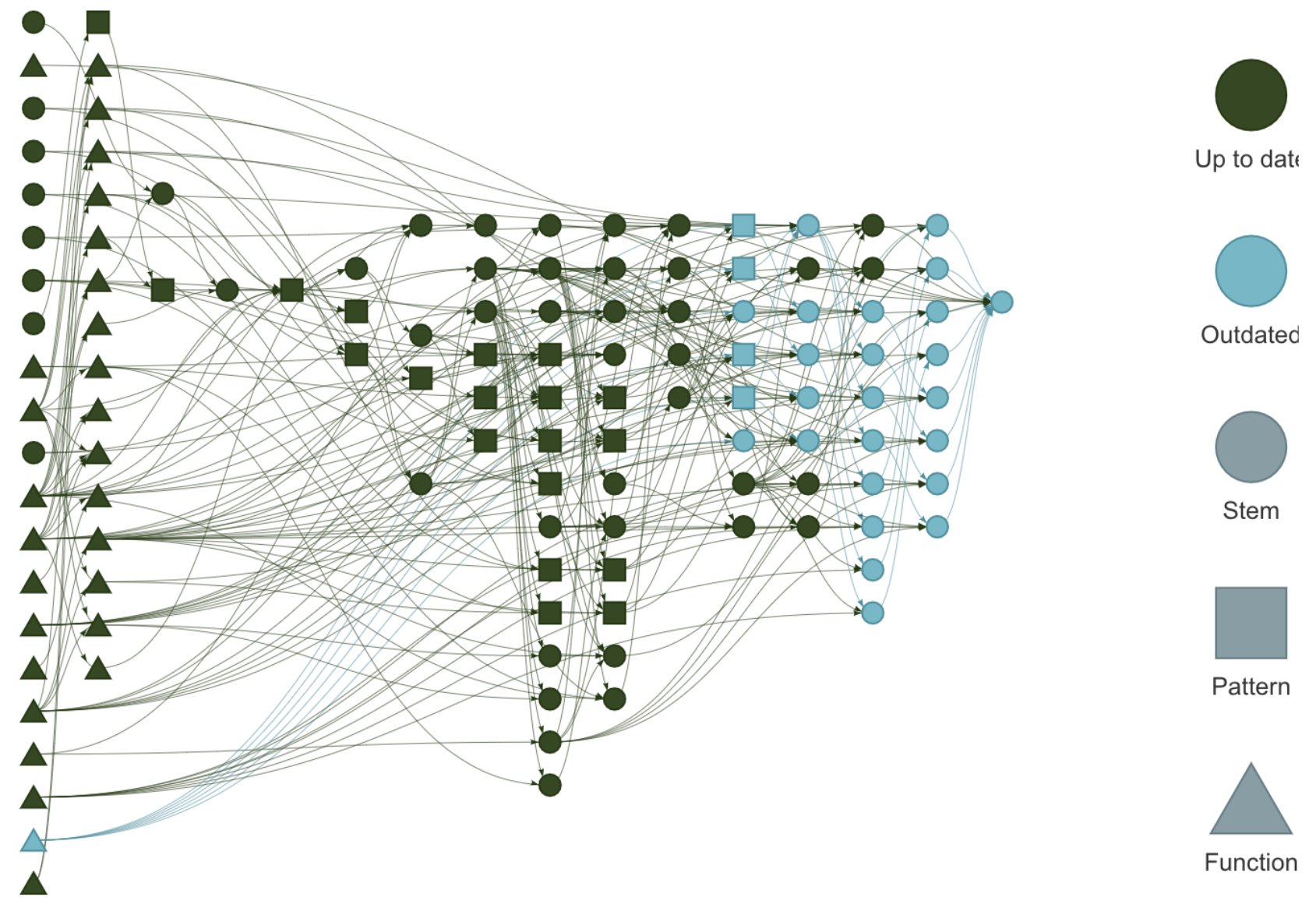
Exemple de graphe de dépendance produit par `targets`

Pipeline à jour



Exemple de graphe de dépendance produit par `targets`

Changement dans une des fonctions



Qualités importantes d'un projet de science des données

Fiabilité

- Confiance dans les résultats*
- Facilité du peer-review*

Exportabilité

- Projet contenu dans un seul répertoire
- Utilisation efficace de la gestion de version (plus pertinent en entreprise)
- Méthode de développement partagée (plus pertinent en entreprise)*

Reproductibilité

- On peut faire rouler le code n'importe quand et obtenir les mêmes résultats*
- N'importe qui peut obtenir les mêmes résultats*
- Pas de problème de version de packages

Confort de développement

- Charge mentale : est-ce que tout est synchronisé, où sont stockés tous les résultats ? Etc.*
- Code modulaire*

***Aspects qui, selon moi, sont significativement améliorés avec `targets`.**

Option #1

```
dossier projet
├── code
│   ├── 01_import_data.R
│   ├── 02_process_data.R
│   ├── 03_models.R
│   └── 04_analysis.R
├── ad_hoc
│   └── report.Rmd
├── data
│   ├── raw
│   │   ├── claims.csv
│   │   └── contracts.csv
│   ├── processed
│   └── results
└── project.Rproj
```

Idée

- On numérote les scripts, donc on sait dans quel ordre les rouler
- Les résultats sont exportés dans le dossier `results`

Problèmes

- Tendance à ajouter encore des étapes : `02_01_pre_modeling.R`
- Faire des longs scripts qui font plusieurs choses en même temps (code pas très modulaire)
- Pas de centralisation des fonctions crées : risque de duplication du code
- Ça devient donc vite difficile à lire

Option #2

```
dossier projet
├── code
│   ├── 01_import_data.R
│   ├── 02_process_data.R
│   ├── 03_models.R
│   └── 04_analysis.R
├── ad_hoc
│   └── report.Rmd
├── R
│   ├── utils.R
│   ├── get_data.R
│   ├── transform_data.R
│   └── fit_model.R
├── data
│   ├── raw
│   │   ├── claims.csv
│   │   └── contracts.csv
│   ├── processed
│   └── results
├── project.Rproj
└── main.R
```

`main.R` ressemble à ça :

```r
purrr::walk(
  .x = list.files("R/", full.names = TRUE),
  .f = source
)

source("code/01_import_data.R")
source("code/02_process_data.R")
source("code/03_models.R")
source("code/04_analysis.R")
```

Idée

- Centralisation des fonctions dans le dossier `R`

Problèmes

- Si on fait un changement quelque part, qu'est-ce qu'il faut rerouler ?
- Est-ce que tout est bien synchronisé (i.e. à jour) ?

Option #3, avec `targets`

```
dossier projet
├── R
│   ├── utils.R
│   ├── get_data.R
│   ├── transform_data.R
│   └── fit_model.R
├── _targets
│   ├── meta
│   └── objects
├── data
│   ├── claims.csv
│   └── contracts.csv
├── ad_hoc
│   └── test_glm_models.R
├── project.Rproj
└── targets.R
```

Définition du pipeline dans `_targets.R`

```r
list(
  tar_target(
    name = data_source_claims,
    command = "data/claims.csv",
    format = "file"
  ),
  tar_target(
    name = data_claims,
    command = get_data(data_source_claims)
  ),
  tar_target(
    name = data_claims_processed,
    command = transform_data(data_claims)
  ),
  tar_target(
    name = claims_model,
    command = fit_model(data_claims_processed)
  )
)
```

Démarrer un nouveau projet `targets`

0. Installer le package : `install.packages("targets")`
1. Créer un nouveau projet `R`
2. Enregistrer un fichier (vide) appelé `_targets.R` dans le dossier de projet
3. Créer 2 nouveaux dossiers : `data` et `R` (recommandé)

```
dossier projet
├── nom_du_projet.Rproj
├── _targets.R
├── data
└── R
```

Fichier `_targets.R`

Dans le script `_targets.R` vous devez :

1. Importer le package : `library(targets)`
2. Importer vos fonctions personnelles (qui sont dans le dossier `R`)
3. Appeller la fonction `tar_option_set()` si vous souhaitez modifier les comportements par défaut
4. Définir vos « targets » avec la fonction `tar_target`. Un target est simplement une étape de votre pipeline. C'est un **objet d'intérêt** dans votre projet. Ça peut être n'importe quel objet `R` : un jeu de données, un graphique, un modèle ajusté, les résultats d'une analyse, etc. Au minimum, un target doit avoir un nom et une commande `R`. Lorsque le pipeline sera exécuté, un dossier `_target/objects` sera créé, dans lequel les objets sont stockés.
5. Le script doit se terminer par une liste (dans n'importe quel ordre) de vos targets

Exemple minimal

```
.
├── exemple_minimal.Rproj
├── _targets.R
├── data
│   └── boston.csv
└── R
    └── make_scatter_plot.R
```

Fichier `make_scatter_plot.R` :

```r
make_scatter_plot ← function(data, x, y) {
  ggplot(data, aes(x = {{x}}, y = {{y}})) +
    geom_point(size = 0.8, alpha = 0.6) +
    theme_bw()
}
```

Fichier `_targets.R` :

```r
library(targets)
library(tidyverse)

walk(fs::dir_ls("R"), source)

list(
  tar_target(boston_file, "data/boston.csv", format = "file"),
  tar_target(boston_data, read_csv(boston_file)),
  tar_target(plot_medv_lstat, make_scatter_plot(boston_data, x = lstat, y = medv)),
  tar_target(lm_fit, lm(medv ~ lstat, data = boston_data)),
  tar_target(lm_pred, predict(lm_fit))
)
```

https://github.com/francisduval/minimal_example_targets

Fonctions populaires du package `targets`

Fonction	Utilité	Habituellement utilisée dans
`tar_target`	définir les targets	`_targets.R`
`tar_make`	exécuter le pipeline	console `R`
`tar_visnetwork`	visualiser le pipeline	console `R`
`tar_manifest`	renvoie une base de données qui décrit le pipeline	console `R`
`tar_option_set`	définir les options	`_targets.R`
`tar_read`	lire les targets	console `R`
`tar_prune`	supprimer les objets dans `_targets/objects` qui ne font plus partie du pipeline	console `R`
`tar_destroy`	supprimer tous les objets (ou une partie des objets, si spécifié en argument) dans `_targets/objects`	console `R`

Exemple minimal – Exercice

1. Télécharger le dépôt sur GitHub et ouvrir le projet `R` sur RStudio
2. Ouvrir le script `_targets.R`. Importer les packages
3. Visualiser le pipeline avec `tar_visnetwork()`
   - Tout est « outdated » car le pipeline n'a pas encore été exécuté
4. Exécuter le pipeline avec `tar_make()`
   - Observer qu'un nouveau dossier `_targets` a été créé
5. Visualiser de nouveau le pipeline avec `tar_visnetwork()`
   - Tout est maintenant à jour
6. Faire une modification de votre choix dans la fonction `make_scatter_plot`
7. Visualiser de nouveau le pipeline avec `tar_visnetwork()`
   - Tous les targets qui dépendent de cette fonction sont maintenant « outdated »
8. Exécuter le pipeline de nouveau avec `tar_make()`
9. Visualiser de nouveau le pipeline avec `tar_visnetwork()` et observer que tout est redevenu à jour
10. Lire un des targets avec la fonction `tar_read`
11. Ajouter un target de votre choix qui dépend d'un ou plusieurs autres targets existants. Visualiser le pipeline. Exécuter le pipeline.

Suivre les modifications de fichiers externes

- Si votre pipeline importe un **fichier de données préexistant** ou crée des fichiers en dehors dossier "`_targets/objects`", il est recommandé de surveiller leurs modifications
- De cette façon, `tar_make()` va mettre à jour automatiquement les targets en aval si ces fichiers changent.

Exemple pour fichier de données externe

Recommandé :

```
list(
  tar_target(boston_file, "data/boston.csv", format = "file"),
  tar_target(boston_data, read_csv(boston_file))
)
```

Non recommandé :

```
list(
  tar_target(boston_data, read_csv("data/boston.csv"))
)
```

Suivre les modifications de fichiers externes

Exporter des graphiques

- Le target suivant va produire un graphique et le stocker dans le magasin de données "_targets/objects" :

```
tar_target(plot_medv_lstat, make_scatter_plot(boston_data, x = lstat, y = medv)),
```

- On peut vouloir à la place exporter ce graphique en format .PNG :

```
tar_target(
  plot_medv_lstat,
  {
    p ← make_scatter_plot(boston_data, x = lstat, y = medv)
    ggsave("mon_graphique.png", plot = p)
    "mon_graphique.png" # Ceci n'est pas obligatoire car ggsave renvoie déjà le nom du path
  },
  format = "file"
),
```

- « `format = "file"` » permet de suivre les modifications faites à l'adresse "`mon_graphique.png`".

Targets de type « pattern »

Considérer l'exemple minimal

- On a un target qui ajuste une régression linéaire simple avec `lstat` comme variable explicative
- On veut maintenant ajuster une régression linéaire simple pour chacune des variables explicatives `lstat`, `crim` et `zn`
- On se définit d'abord une fonction qui ajuste une régression linéaire simple :

```
fit_simple_lm ← function(data, x, y) {
  lm(formula(paste(y, "~", x)), data = data)
}
```

- Ensuite, on définit le target des variables explicatives :

```
tar_target(var_vec, c("lstat", "crim", "zn")),
```

- Finalement, on ajuste les régressions linéaires et on les stocke dans une liste :

```
tar_target(lm_fit_list, map(var_vec, ~ fit_simple_lm(boston_data, x = ., y = "medv"))),
```

- Problème : si on ajoute un élément à `var_vec`, tout sera re-roulé (c'est problématique si le target `lm_fit_list` est long à rouler)

Targets de type « pattern »

- Une meilleure alternative est de créer un target de type « pattern » :

```
tar_target(lm_fit_pattern, fit_simple_lm(boston_data, x = var_vec, y = "medv"), pattern = map(var_vec)),
```

- `targets` va alors créer un target possédant plusieurs « branches ».
- Cette manière de procéder est appelée **dynamic branching**

Exercice :

1. Ajouter le target `var_vec` ainsi que le pattern `lm_fit_pattern` à votre pipeline
2. Rouler le pipeline et l'examiner avec `tar_visnetwork`
3. Importer le target `lm_fit_pattern` dans l'environnement global avec `tar_read` et l'examiner
4. Ajouter une variable au target `var_vec`
5. Visualiser le pipeline
6. Rouler le pipeline de nouveau. Observer que `targets` va sauter les branches qui ont déjà été roulées

Défi pour après la présentation :

- Définir un target de type « pattern » qui prend en entrée le pattern `lm_fit_pattern` (entre autres) et qui renvoie un pattern contenant le nuage de points et la droite de régression pour chacune des régressions

Fichier RMarkdown en tant que target

- Il est possible avec la fonction `tarchetypes::tar_render()` d'ajouter un **rapport RMarkdown** à votre pipeline
- Votre rapport devrait **s'exécuter rapidement** : les gros calculs devraient être faits dans les autres targets

Intégrer un rapport RMarkdown à votre pipeline :

1. Écrire votre rapport dans un fichier `.Rmd`. Celui-ci devrait importer des targets avec la fonction `tar_read()`. Le sauvegarder dans le répertoire
2. Ajouter le target du rapport à votre pipeline avec `tarchetypes::tar_render()` :

```
tar_render(rapport, "rapport.Rmd"),
```

3. Appeler `tar_visnetwork()` pour voir si le rapport est bel et bien intégré
4. Rouler le pipeline et observer que votre rapport (en format .html ou .pdf) est apparu dans le répertoire

Options

- Il est possible d'écraser plusieurs comportements par défaut avec la fonction `tar_option_set`
- À mettre au début du script `_targets.R` :

```r
tar_option_set(
  garbage_collection = T, # je fixe toujours à TRUE
  memory = "transient", # je fixe toujours à "transient"
  format = "qs", # je fixe toujours à "qs"
  workspace_on_error = F, # je laisse toujours à FALSE sauf pour débogger
  iteration = "list", # autre option: "vector"
  packages = c("tidyverse", "ggplot") # pacakges utilisés
)
```

- Certaines options peuvent être changées dans chaque target de manière individuelle

Vive les fonctions !

3 bénéfices de créer des fonctions

- Meilleure abstraction du code
- Pas de duplication du code (on peut réutiliser les fonctions)
- Le fichier `_targets.R` est plus facile à lire et moins long avec des fonctions

Recommandé :

```
tar_target(plot_medv_lstat, make_scatter_plot(boston_data, x = lstat, y = medv)),
```

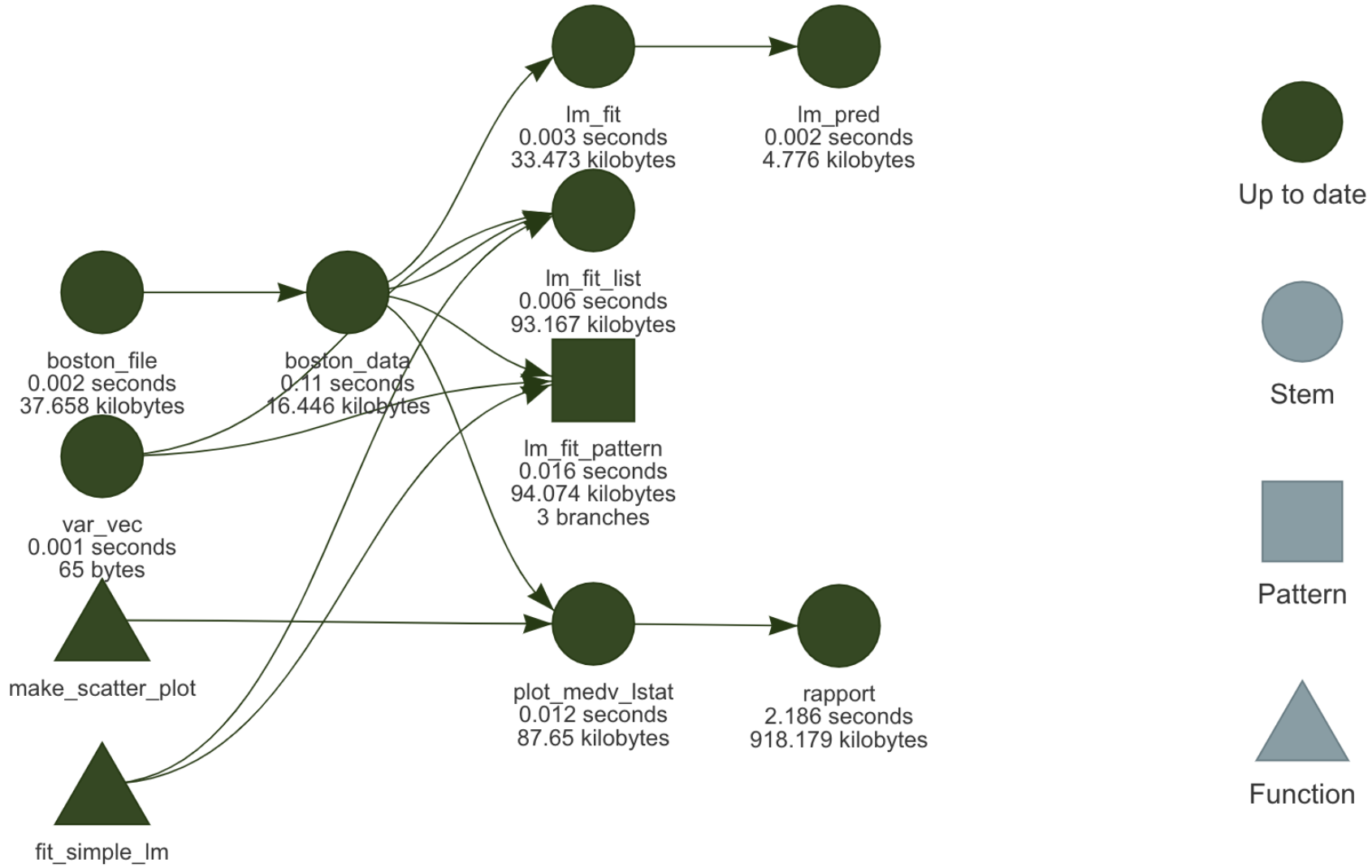
Non recommandé :

```
tar_target(
  plot_medv_lstat,
  ggplot(boston_data, aes(x = lstat, y = medv)) +
    geom_point(size = 0.9, alpha = 0.6) +
    theme_bw()
)
```

Quelques trucs utiles

Plus de détails avec `tar_visnetwork`

```
tar_visnetwork(label = c("time", "size", "branches"))
```



Quelques trucs utiles

Empêcher un target de s'exécuter

```
tar_target(
  lm_pred,
  predict(lm_fit),
  cue = tar_cue(mode = "never")
),
```

Liens utiles

- **Présentation du package par Will Landau :**
  - `https://www.youtube.com/watch?v=Gqn7Xn4d5NI&t`
- **Excellent tutoriel d'une demi-journée créé par Will Landau (demande d'avoir un compte RStudio Cloud) :**
  - `https://rstudio.cloud/project/1699460`
- **Manuel de l'utilisateur du package :**
  - `https://books.ropensci.org/targets/`

Exercice

Sélectionnez un jeu de données quelconque. Construire un pipeline de science des données à partir de ce jeu de données. Votre projet devra :

- suivre les modifications du jeu de données choisi (en utilisant « `format = "file"` »),
- contenir un target qui exporte un ou plusieurs graphique(s) dans un dossier nommé « figures » (que vous devez créer),
- contenir au moins un target de type « pattern »,
- contenir un target qui suit les modifications d'un rapport RMarkdown. Vous devez préalablement créer ce rapport, que vous sauvegarderez dans un dossier nommé « reports ».