How to use Fast Step Graph

Juan G. Colonna* Marcelo Ruiz†

To install the last version of this package directly from GitHub uncomment and run:

```
# library(devtools)
# use "quiet = FALSE" if you want to see the outputs of this command
# devtools::install_github("juancolonna/FastStepGraph", quiet = TRUE, force = TRUE)

# Then, load it:
library(FastStepGraph)

# If you directly cloned the github repository,
# then you should uncomment these lines to load the functions:
# source('FastStepGraph.R')
# source('SigmaAR.R')
```

Simulate Gaussian Data with an Autoregressive (AR) Model:

```
set.seed(1234567)
phi <- 0.4
p <- 50  # number of variables (dimension)
n <- 30 # number of samples

# Generate Data from a Gaussian distribution
data <- FastStepGraph::SigmaAR(n, p, phi)
X <- scale(data$X) # standardizing variables
```

Afterwards, fit the Omega matrix by calling the Fast Step Graph function, like:

```
t0 <- Sys.time() # INITIAL TIME
G <- FastStepGraph::FastStepGraph(X, alpha_f = 0.22, alpha_b = 0.14)
difftime(Sys.time(), t0, units = "secs")
#> Time difference of 0.08767033 secs
# print(G$Omega)
```

If the `nei.max` argument is omitted, it will be 5. If you don't know the `alpha_f` and `alpha_b` values, the use cross-validation. If your input variables are non-standardized (with zero mean and unit variance), we recommend that you set `data_scale=TURE`. To find the optimal $\alpha_{\mathbf{f}}$ and $\alpha_{\mathbf{b}}$ parameters for the previously generated $\mathbf{X}$ data, we can perform a cross-validation on a combination grid as follows:

```
t0 <- Sys.time() # INITIAL TIME
res <- FastStepGraph::cv.FastStepGraph(X, data_shuffle = TRUE)
difftime(Sys.time(), t0, units = "secs")
#> Time difference of 4.261183 secs

# print(res$alpha_f_opt)
```

*Institute of Computing. Federal University of Amazonas. Brasil. juancolonna@icomp.ufam.edu.br
†Mathematics Department. National University of Río Cuarto. Argentina. mruiz@exa.unrc.edu.ar

```r
# print(res$alpha_b_opt)
# print(res$Omega)
```

The arguments `n_folds = 5`, `alpha_f_min = 0.1`, `alpha_f_max = 0.9`, `n_alpha = 32` (size of the grid search) and `nei.max = 5`, have defaults values and can be omitted. Note that, `cv.FastStepGraph(X)` is not an exhaustive grid search over $\alpha_{\mathbf{f}}$ and $\alpha_{\mathbf{b}}$. This is a heuristic that always sets $\alpha_{\mathbf{b}} = \frac{\alpha_{\mathbf{f}}}{2}$. It is recommended to shuffle the rows of `X` before running cross-validation. The default value is `data_shuffle = TRUE`, but if you want to disable row shuffle, set it to `data_shuffle = FALSE`.

To increase time performance, you can run `cv.FastStepGraph(X, parallel = TRUE)` in parallel. Before, you'll need to install and register a parallel backend. To run on a Linux system the **doParallel** dependency must be installed `install.packages("doParallel")`. These parallel packages will also require the following dependencies: **foreach**, **iterators** and **parallel**. Make sure you satisfy them. Then, call the method setting the parameter **parallel = TRUE**, as follows:

```r
t0 <- Sys.time() # INITIAL TIME
# use 'n_cores = NULL' to set the maximum number of cores minus one on your machine
res <- FastStepGraph::cv.FastStepGraph(X, parallel = TRUE, n_cores = 2)
difftime(Sys.time(), t0, units = "secs")

# print(res$alpha_f_opt)
# print(res$alpha_b_opt)
# print(res$Omega)
```

Remember, you can set the `n_cores` parameter to a value equal to the number of cores you have, but be careful as this may freeze your system. Setting it to 1 disables parallel processing, and setting it to a number greater than the number of available cores does not improve efficiency.