Getting started with `hierNetGxE`

Natalia Zemlianskaia

2021-01-27

Purpose of this vignette

This vignette is a tutorial about how to use the `hierNetGxE` package

- for the selection of the gene-environment interaction terms in a joint "main-effect-before-interaction" hierarchical manner
- for building a *correctly spesified* prediction model containing interaction terms with a specific exposure of interest, where the final prediction model only includes interaction terms for which their respective main effects are also included in the model
- to show how to convert data to sparse/bigmatrix format for more efficient and flexible use of the package

Overview

The package is developed to fit a regularized regression model that we call **hierNetGxE** for the joint selection of gene-environment (GxE) interactions based on the hierarchical lasso [Bien et al. (2013)]. The model focuses on a single environmental exposure and induces a "main-effect-before-interaction" hierarchical structure. Unlike the original hierarchical lasso model, which was designed for the gene-gene (GxG) interaction case, the GxE model has a simpler block-separable structure that makes it possible to fit in large-scale applications. We developed and implemented an efficient fitting algorithm and screening rules that can discard large numbers of irrelevant predictors with high accuracy.

hierNetGxE model induces hierarchical selection of the (GxE) interaction terms via convex constraints added to the objective function. The model has two tuning parameters λ_1 and λ_2 responsible for the model sparsity with respect to main effects and interactions respectively.

Response (outcome) variable can be either quantitative of binary 0/1 variable. For a binary response IRLS procedure is used with custom screening rules we developed.

Installation

Installation can take a couple of minutes because of the requirement to install dependent packages (`dplyr`, `Rcpp`, `RcppEigen`, `RcppThread`, `BH`, `bigmemory`)

```
## install.packages("devtools")

library(devtools)
devtools::install_github("NataliaZemlianskaia/hierNetGxE")
```

Attaching libraries for the examples below

```
library(hierNetGxE)
library(glmnet)
library(ggplot2)
```

Data generation

First, we generate the dataset using `data.gen()` function from the package. The function allows us to generate a binary (0/1) matrix of genotypes `G` of a given `sample_size` and `p` number of features, binary vector `E` of environmental measurements, and a response variable `Y` $\sim g(\beta_0 + \beta_E E + \beta_G G + \beta_{G\times E} G \times E$ (either quantitative or binary depending of the `family` parameter) .

We can specify the number of non-zero main effects (`n_g_non_zero`) and interactions (`n_gxe_non_zero`) we want to generate. We also specified a *"strong_hierarchical"* mode for our dataset, which means that (1) if interaction effect is generated as non-zero, it's respective genetic main effect is also generated as non-zero ($\beta_{G\times E} \neq 0 \longrightarrow \beta_G \neq 0$), and (2) effect sizes of the main effects are larger than that of interaction effects ($|\beta_G| \geq |\beta_{G\times E}|$).

```
family = "gaussian"
sample_size = 150; p = 400; n_g_non_zero = 10; n_gxe_non_zero = 5

data = data.gen(seed=1, sample_size=sample_size, p=p,
                n_g_non_zero=n_g_non_zero, n_gxe_non_zero=n_gxe_non_zero,
                mode="strong_hierarchical",
                family=family)
```

Let's look at the dataset we generated `data$G_train`, `data$E_train`, and `data$Y_train`

```
dim(data$G_train)
```

```
## [1] 150 400
```

```
head(data$G_train[,1:5])
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    1    0    0
## [3,]    0    0    0    0    0
## [4,]    1    0    1    1    0
## [5,]    0    1    0    0    0
## [6,]    1    0    0    0    0
```

```
data$E_train[1:10]
```

```
##  [1] 0 0 0 0 0 0 0 0 1 1
```

```
head(data$Y_train)
```

```
## [1]  2.1951026 -0.2859079 -2.8515097 -2.6861760 -5.9169406  2.0882565
```

We generated model coefficients `data$Beta_G` and `data$Beta_GxE` such that we got `sum(data$Beta_G != 0)` = 10 non-zero main effects and `sum(data$Beta_GxE != 0)` = 5 non-zero interaction effects as we specified.

```
c(sum(data$Beta_G != 0), sum(data$Beta_GxE != 0))
```

```
## [1] 10  5
```

We also imposed *strong_hierarchical* relationships between main and interaction effect sizes

```
cbind(data$Beta_G[data$Beta_G != 0], data$Beta_GxE[data$Beta_G != 0])
```

```
##       [,1] [,2]
##  [1,]   -3  0.0
##  [2,]   -3  1.5
##  [3,]   -3 -1.5
##  [4,]    3  0.0
##  [5,]   -3  0.0
##  [6,]    3 -1.5
##  [7,]    3  0.0
##  [8,]   -3  0.0
##  [9,]   -3 -1.5
## [10,]   -3  1.5
```

Selection example

To tune the model over a 2D grid of hyper-parameters (lambda_1 and lambda_2) we use function `hierNetGxE.cv()`, where we specify

- our data `G, E, Y` and `family` parameter for the response variable; matrix `G` should be a numeric matrix, vector `E` should be a numeric vector. Vector `Y` could either be a numeric or binary 0/1 vector
- `tolerance` for the convergence of the fitting algorithm
- `grid_size` to automatically generate grid values for cross-validation
- `nfolds` to specify how many folds we want to use in cross-validation
- `parallel` TRUE to enable parallel cross-validation
- `seed` set random seed to control random folds assignments
- `normalize` TRUE to normalize matrix `G` (such that column-wise sd is eqial to 1)

```
tolerance = 1e-4
start = Sys.time()
tune_model = hierNetGxE.cv(G=data$G_train, E=data$E_train, Y=data$Y_train,
                           family=family, grid_size=20, tolerance=tolerance,
                           parallel=TRUE, nfold=3,
                           normalize=TRUE,
                           seed=1)
```

```
## [1] "Parallel cv"
## Time difference of 8.149458 secs
## [1] "fit on full data"
## Time difference of 9.883202 secs
```

```
Sys.time() - start
```

```
## Time difference of 18.05552 secs
```

Obtain best model with `hierNetGxE.coef()` function

To obtain interaction and main effect coefficients corresponding to the model with a particular pair of tuning parameters using `hierNetGxE.coef()` function. We need to specify a model fit object and a pair of lambda = (lambda_1, lambda_2) values organized in a tibble (ex: `lambda = tibble(lambda_1=tune_model$grid[1], lambda_2=tune_model$grid[1])`).

Here we set `fit=tune_model$fit` - model fit on the full dataset and `lambda=tune_model$lambda_min` - the pair of tuning parameters corresponding to the minimum cross-validation error that `hierNetGxE.cv()` function returns.

```
coefficients = hierNetGxE.coef(fit=tune_model$fit, lambda=tune_model$lambda_min)
gxe_coefficients = coefficients$beta_gxe
g_coefficients = coefficients$beta_g
```

Check if all non-zero interaction features were recovered by the model

```
cbind(data$Beta_GxE[data$Beta_GxE != 0], gxe_coefficients[data$Beta_GxE != 0])
```

```
##      [,1]       [,2]
## [1,]  1.5  0.8485495
## [2,] -1.5 -1.8750650
## [3,] -1.5 -0.7304513
## [4,] -1.5 -1.8583926
## [5,]  1.5  0.9657081
```

Check that the largest estimated interaction effect sizes correspond to the true non-zero coefficients

```
(data$Beta_GxE[order(abs(gxe_coefficients), decreasing=TRUE)])[1:10]
```

```
##  [1] -1.5 -1.5  1.5  1.5 -1.5  0.0  0.0  0.0  0.0  0.0
```

Calculate principal selection metrics with `selection.metrics()` funciton available in the package

```
selection_hierNetGxE = selection.metrics(true_b_g=data$Beta_G, true_b_gxe=data$Beta_GxE,
                                         estimated_b_g=g_coefficients, estimated_b_gxe=gxe_coefficients)
cbind(selection_hierNetGxE)
```

```
##                 selection_hierNetGxE
## b_g_non_zero    45
## b_gxe_non_zero  32
## mse_b_g         0.319372
## mse_b_gxe       0.5605675
## sensitivity_g   1
```

```
## specificity_g   0.9102564
## precision_g     0.2222222
## sensitivity_gxe 1
## specificity_gxe 0.9316456
## precision_gxe   0.15625
## auc_g           0.999999
## auc_gxe         0.999998
```

Compare with the standard Lasso model (we use `glmnet` package)

```
set.seed(1)
tune_model_glmnet = cv.glmnet(x=cbind(data$E_train, data$G_train, data$G_train *
  data$E_train),
                              y=data$Y_train,
                              nfolds=3,
                              family=family)

coef_glmnet = coef(tune_model_glmnet, s=tune_model_glmnet$lambda.min)
g_glmnet = coef_glmnet[3: (p + 2)]
gxe_glmnet = coef_glmnet[(p + 3): (2 * p + 2)]

cbind(data$Beta_GxE[data$Beta_GxE != 0], gxe_glmnet[data$Beta_GxE != 0])
```

```
##      [,1]      [,2]
## [1,]  1.5 0.0000000
## [2,] -1.5 0.0000000
## [3,] -1.5 0.0000000
## [4,] -1.5 0.0000000
## [5,]  1.5 0.9937375
```

```
(data$Beta_GxE[order(abs(gxe_glmnet), decreasing=TRUE)])[1:10]
```

```
##  [1] 1.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

```
selection_glmnet = selection.metrics(data$Beta_G, data$Beta_GxE, g_glmnet, gxe_glmnet)
cbind(selection_hierNetGxE, selection_glmnet)
```

```
##                 selection_hierNetGxE selection_glmnet
## b_g_non_zero    45                   50
## b_gxe_non_zero  32                   25
## mse_b_g         0.319372             0.2532511
## mse_b_gxe       0.5605675            1.36061
## sensitivity_g   1                    1
## specificity_g   0.9102564            0.8974359
## precision_g     0.2222222            0.2
## sensitivity_gxe 1                    0.2
## specificity_gxe 0.9316456            0.9392405
## precision_gxe   0.15625              0.04
```

```
## auc_g           0.999999            0.999999
## auc_gxe         0.999998            0.6146823
```

We see that hierNetGxE model outperforms the Lasso model in terms of selection of interaction terms in all important selection metrics, including GxE detection AUC (`auc_gxe`), sensitivity (`sensitivity_gxe`), and precision(`precision_gxe`). Estimation bias of the interaction coefficients is also lower for the hierNetGxE model (`mse_b_gxe`).

Obtain target model with `hierNetGxE.coefnum()` function

To control how parsimonious is our model with respect to the selected non-zero interaction terms we can use `hierNetGxE.coefnum()` function. We obtain interaction and main effect coefficients corresponding to the target GxE model, where instead of specifying tuning parameters `lambda` we set `target_b_gxe_non_zero` parameter which is the number of at most non-zero interactions we want to include in the model.

```
coefficients = hierNetGxE.coefnum(cv_model=tune_model, target_b_gxe_non_zero=10)
gxe_coefficients = coefficients$beta_gxe
g_coefficients = coefficients$beta_g
```

Calculate principal selection metrics

```
selection_hierNetGxE = selection.metrics(data$Beta_G, data$Beta_GxE, g_coefficients,
  gxe_coefficients)
cbind(selection_hierNetGxE, selection_glmnet)
```

```
##                 selection_hierNetGxE selection_glmnet
## b_g_non_zero    21                   50
## b_gxe_non_zero  8                    25
## mse_b_g         0.4366771            0.2532511
## mse_b_gxe       1.043102             1.36061
## sensitivity_g   1                    1
## specificity_g   0.9717949            0.8974359
## precision_g     0.4761905            0.2
## sensitivity_gxe 0.8                  0.2
## specificity_gxe 0.9898734            0.9392405
## precision_gxe   0.5                  0.04
## auc_g           0.999999             0.999999
## auc_gxe         0.9665803            0.6146823
```

Here we see that the number of non-zero interactions in the model (`b_gxe_non_zero`) is 8 (vs 32 non-zero terms in the model corresponding to the minimal cross-validated error), leading to increased precision and slightly decreased sensitivity.

Prediction example

With `hierNetGxE.coef()` we, again, obtain coefficients corresponding to the best model in terms of minimal cross-validated error and use `hierNetGxE.predict()` function to apply our estimated model to a new test dataset that can also be generated by the function `data.gen()`.

We calculate test R-squared to compare model performance.

```
coefficients = hierNetGxE.coef(tune_model$fit, tune_model$lambda_min)
beta_0 = coefficients$beta_0; beta_e = coefficients$beta_e
beta_g = coefficients$beta_g; beta_gxe = coefficients$beta_gxe

new_G = data$G_test; new_E = data$E_test
new_Y = hierNetGxE.predict(beta_0, beta_e, beta_g, beta_gxe, new_G, new_E)
test_R2_hierNetGxE = cor(new_Y, data$Y_test)^2
```

Compare with the standard Lasso (we use `glmnet` package)

```
new_Y_glmnet = predict(tune_model_glmnet, newx=cbind(new_E, new_G, new_G * new_E),
                       s=tune_model_glmnet$lambda.min)
test_R2_glmnet = cor(new_Y_glmnet[,1], data$Y_test)^2
cbind(test_R2_hierNetGxE, test_R2_glmnet)
```

```
##      test_R2_hierNetGxE test_R2_glmnet
## [1,]          0.9773027      0.9286629
```

Sparse matrix option example

We generate large highly sparse genotype matrix `G` by setting genotype prevalence parameter `pG` to 0.03 and measure how long does it take to fit the model.

```
sample_size = 600; p = 30000
pG = 0.03
data = data.gen(seed=1, sample_size=sample_size, p=p,
                n_g_non_zero=n_g_non_zero, n_gxe_non_zero=n_gxe_non_zero,
                mode = "strong_hierarchical",
                pG=pG,
                family=family)
sum(data$G_train != 0) / (sample_size * p)
```

```
## [1] 0.03005317
```

```
start = Sys.time()
fit = hierNetGxE.fit(G=data$G_train, E=data$E_train, Y=data$Y_train,
                     tolerance=tolerance,
                     normalize=TRUE)
time_non_sparse = Sys.time() - start; time_non_sparse
```

```
## Time difference of 1.273677 mins
```

We then convert our generated matrix to sparse format and measure fitting time again.

```
G_train_sparse = as(data$G_train, "dgCMatrix")

start = Sys.time()
```

```
fit = hierNetGxE.fit(G=G_train_sparse, E=data$E_train, Y=data$Y_train,
                     tolerance=tolerance,
                     normalize=TRUE)
time_sparse = Sys.time() - start; time_sparse
```

```
## Time difference of 40.62844 secs
```

We see that sparse option enables more efficient fitting, however, the speed-up works only for substantially sparse matrices.

```
(as.numeric(time_non_sparse) * 60) / as.numeric(time_sparse)
```

```
## [1] 1.880964
```

Bigmatrix (`bigmemory` package) option example

Here we show how to convert data into `bigmatrix` format. This format should be used for a very large datasets that barely fit in RAM.

For out of RAM objects function `attach.big.matrix("g_train.desc")` should be used given that the files already exist in a correct format (`g_train.desc`, `g_train.bin`).

```
grid = compute.grid(data$G_train, data$E_train, data$Y_train,
                    normalize=TRUE, grid_size=grid_size, grid_min_ratio=1e-4)

G_train = as.big.matrix(data$G_train,
                        backingfile="g_train.bin",
                        descriptorfile="g_train.desc")
is.filebacked(G_train)

fit_bm = hierNetGxE.fit(G_train, data$E_train, data$Y_train,
                     tolerance=tolerance, grid=grid, family=family)

tune_model_bm = hierNetGxE.cv(G_train, data$E_train, data$Y_train,
                              family=family, grid=grid, tolerance=tolerance,
                              parallel=TRUE, nfold=3,
                              normalize=TRUE, seed=1)
```