Getting started with `gesso`

Natalia Zemlianskaia

2021-04-14

Purpose of this vignette

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

- for the selection of the gene-environment interaction terms in a joint *main-effect-before-interaction* hierarchical manner
- for building a *well-formulated* 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

Overview

The package is developed to fit a regularized regression model that we call **gesso** for the joint selection of gene-environment (GxE) interactions. The model focuses on a single environmental exposure and induces a *main-effect-before-interaction* hierarchical structure. We developed and implemented an efficient fitting algorithm and screening rules that can discard large numbers of irrelevant predictors with high accuracy.

gesso model induces hierarchical selection of the (GxE) interaction terms via overlapped group lasso structure. The model has two tuning parameters λ_1 and λ_2 allowing flexible, data-dependent control over group sparsity and additional interactions sparsity.

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

The package supports sparse matrices `dgCMatrix` and (filebacked) bigmatrix format from the `bigmemory` package for large or out of RAM datasets.

Package highlights

The package allows

- to tune only one hyperparameter (λ_1) and set α: $\lambda_2 = \alpha\ \lambda_1$;
- to tune hyperparameters by maximizing AUC for the binary outcomes;
- to add unpenalized covariates to the model;
- to utilize sparse/bigmatrix data formats for more efficient and flexible use of the package.

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/gesso")
```

Attaching libraries for the examples below

```
library(gesso)
library(glmnet)
library(ggplot2)
library(bigmemory)
```

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

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

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

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

```
## [1]  5.6849366  0.6601472  8.6602417  2.9610805  8.2804916 -2.3606432
```

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 effects and interaction effects

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

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

Selection example

To tune the model over a 2D grid of hyper-parameters (`lambda_1` and `lambda_2`) we use the function `gesso.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 continuous or binary 0/1 numeric 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 to use in cross-validation
- `parallel` TRUE to enable parallel cross-validation (TRUE by default)
- `seed` set the random seed to control random folds assignments
- `normalize` TRUE to normalize matrix `G` (such that column-wise sd is eqial to 1) and vector `E` (TRUE by default)
- `normalize_response` TRUE to normalize vector `Y` (FALSE by default)

There is an option to use a custom grid in `gesso.cv()` function by specifying `grid` parameter instead of `grid_size` parameter.

`verbose` parameter is set to TRUE by default in `gesso.cv()`, but we can set it to FALSE to avoid outputting function messages about partial execution time.

```
start = Sys.time()
tune_model = gesso.cv(G=data$G_train, E=data$E_train, Y=data$Y_train,
                      family=family, grid_size=20, tolerance=1e-4,
                      parallel=TRUE, nfolds=3,
                      normalize=TRUE,
                      normalize_response=TRUE,
                      seed=1,
                      max_iterations=10000)
```

```
## Compute grid:
## Time difference of 0.001646996 secs
## Parallel cv:
## Time difference of 1.07461 secs
## Fit on the full dataset:
## Time difference of 0.7625861 secs
```

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

```
## Time difference of 1.850275 secs
```

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

To obtain interaction and main effect coefficients corresponding to the model with a particular pair of tuning parameters we use `gesso.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])`).

Below 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 `gesso.cv()` function returns.

```
coefficients = gesso.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 -0.2166961
## [2,] -1.5 -1.6072629
## [3,] -1.5  0.0000000
## [4,] -1.5 -0.7346621
## [5,]  1.5  1.2926505
```

Check that the largest estimated interaction effects 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  0.0 -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_gesso = 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_gesso)
```

```
##                 selection_gesso
## b_g_non_zero    51
## b_gxe_non_zero  16
## mse_b_g         0.3235779
## mse_b_gxe       0.952587
## sensitivity_g   1
## specificity_g   0.8948718
## precision_g     0.1960784
## sensitivity_gxe 0.8
## specificity_gxe 0.9696203
## precision_gxe   0.25
## auc_g           1
## auc_gxe         0.8964557
```

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 -0.0224555
## [3,] -1.5  0.0000000
## [4,] -1.5  0.0000000
## [5,]  1.5  1.2517060
```

```
(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_gesso, selection_glmnet)
```

```
##                 selection_gesso selection_glmnet
## b_g_non_zero    51              62
## b_gxe_non_zero  16              26
## mse_b_g         0.3235779       0.2966215
## mse_b_gxe       0.952587        1.341252
## sensitivity_g   1               1
## specificity_g   0.8948718       0.8666667
## precision_g     0.1960784       0.1612903
## sensitivity_gxe 0.8             0.4
## specificity_gxe 0.9696203       0.9392405
## precision_gxe   0.25            0.07692308
## auc_g           1               1
## auc_gxe         0.8964557       0.6711392
```

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

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

To control how parsimonious is our model with respect to the selected non-zero interaction terms we can use `gesso.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* (specify `less_than=TRUE` - default value) or *at least* (specify `less_than=FALSE`) non-zero interactions we want to include in the model, depending on the parameter `less_than`.

The target model is selected based on the averaged cross-validation (cv) results (tune_model$cv_result): for each pair of parameters `lambda`=(`lambda_1`, `lambda_2`) in the grid and for each cv fold we obtain a number of non-zero estimated interaction terms, then average cv results by lambda and choose the tuning parameters corresponding to the minimum average cv loss that have *at most* or *at least* `target_b_gxe_non_zero` non-zero interaction terms. Returned coefficients are obtained by fitting the model on the full dataset with the selected tuning parameters.

Note that the number of estimated non-zero interactions will only approximately reflect the numbers obtained on cv datasets.

```
coefficients = gesso.coefnum(cv_model=tune_model, target_b_gxe_non_zero=5)
gxe_coefficients = coefficients$beta_gxe
g_coefficients = coefficients$beta_g
```

Calculate principal selection metrics

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

```
##                 selection_gesso selection_glmnet
```

```
## b_g_non_zero    61              62
## b_gxe_non_zero  5               26
## mse_b_g         0.3564593       0.2966215
## mse_b_gxe       1.134043        1.341252
## sensitivity_g   1               1
## specificity_g   0.8692308       0.8666667
## precision_g     0.1639344       0.1612903
## sensitivity_gxe 0.6             0.4
## specificity_gxe 0.9949367       0.9392405
## precision_gxe   0.6             0.07692308
## auc_g           1               1
## auc_gxe         0.7989873       0.6711392
```

Here we see that the number of non-zero interactions in the model (`b_gxe_non_zero`) is 4 (vs 45 non-zero terms in the model corresponding to the minimal cross-validated error), leading to substantially increased precision and decreased sensitivity.

Prediction example

With `gesso.coef()` we obtain coefficients corresponding to the best model in terms of minimal cross-validated error and use `gesso.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 assess model performance.

```
coefficients = gesso.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 = gesso.predict(beta_0, beta_e, beta_g, beta_gxe, new_G, new_E)
test_R2_gesso = 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_gesso, test_R2_glmnet)
```

```
##      test_R2_gesso test_R2_glmnet
## [1,]     0.9694657      0.9437631
```

Adding unpenalized covariates to the model

The package allows adding unpenalized covariates to the model (for example, important adjustment demographic variables like age and gender). In this example, we first generate data with additional covariates (specify `n_confounders` parameter in `data.gen()`) and then show how to fit the model, the user just needs to specify numeric matrix `C` of covariates organized by columns.

```
family = "gaussian"
sample_size = 150; p = 400; n_g_non_zero = 10; n_gxe_non_zero = 5
n_confounders = 2

grid = 10^seq(-3, log10(1), length.out = 20)

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,
                n_confounders=n_confounders)

tune_model = gesso.cv(G=data$G_train, E=data$E_train, Y=data$Y_train,
                      C=data$C_train,
                      family=family, grid=grid, tolerance=1e-4,
                      parallel=TRUE, nfolds=3,
                      normalize=TRUE,
                      normalize_response=TRUE,
                      verbose=FALSE,
                      seed=1)
```

Sparse matrix option example

We generate a large highly sparse genotype matrix `G` by setting the 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] 0.03003111
```

```
start = Sys.time()
fit = gesso.fit(G=data$G_train, E=data$E_train, Y=data$Y_train,
                grid_size=20, grid_min_ratio=1e-2,
                tolerance=1e-4,
                normalize=TRUE,
                normalize_response=TRUE)
time_non_sparse = Sys.time() - start; time_non_sparse
```

```
## Time difference of 1.129352 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 = gesso.fit(G=G_train_sparse, E=data$E_train, Y=data$Y_train,
                tolerance=1e-4,
                grid_size=20, grid_min_ratio=1e-2,
                normalize=TRUE,
                normalize_response=TRUE)
time_sparse = Sys.time() - start; time_sparse
```

```
## Time difference of 18.14016 secs
```

We see that the sparse option enables (0.06 times) more efficient fitting, however, the speed-up works only for substantially sparse matrices.

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

```
## [1] 0.06225699
```

Bigmatrix (bigmemory package) option example

Here we show how to convert data into `bigmatrix` format. This format should be used for 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`).

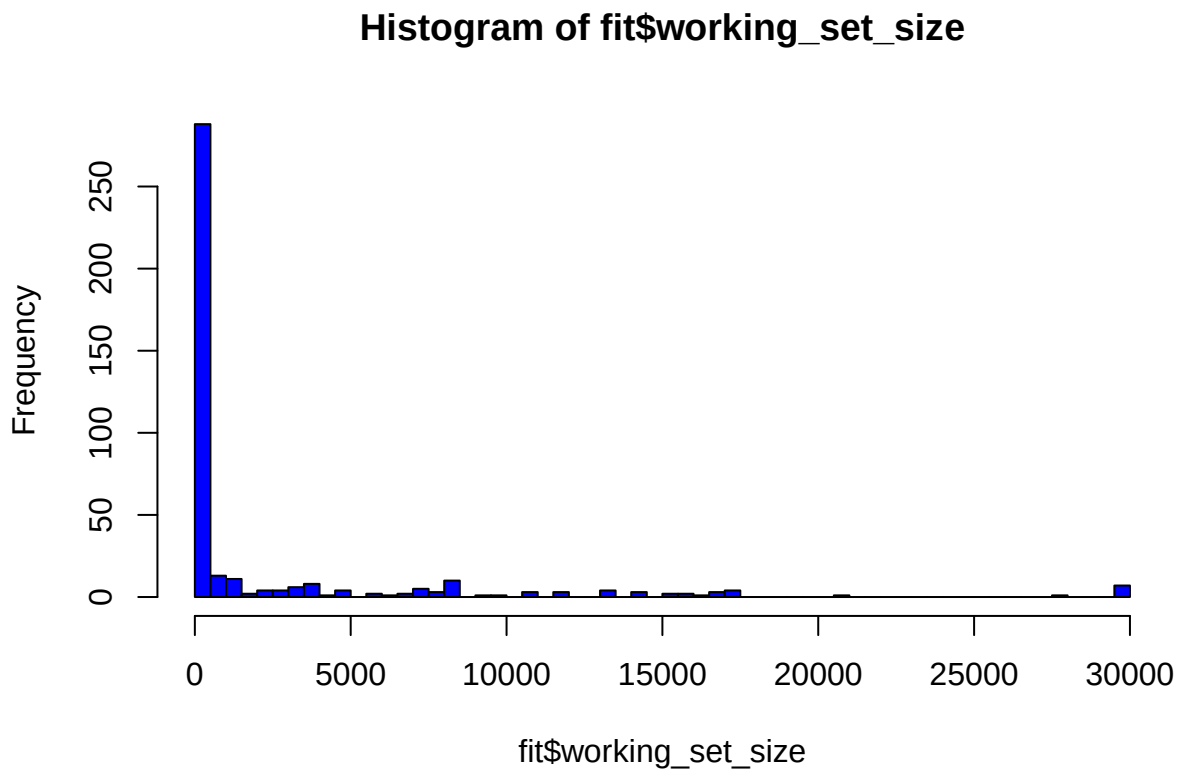
```
grid = compute.grid(data$G_train, data$E_train, data$Y_train,
                    normalize=TRUE, grid_size=grid_size, grid_min_ratio=1e-4)

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

fit_bm = gesso.fit(G_train, data$E_train, data$Y_train,
                   tolerance=1e-4, grid=grid, family=family)

tune_model_bm = gesso.cv(G_train, data$E_train, data$Y_train,
                         family=family, grid=grid, tolerance=1e-4,
                         parallel=TRUE, nfolds=3,
                         normalize=TRUE, seed=1)
```

Working sets

We use the same large dataset ($p = 30{,}000$) that we generated for the sparse matrices example to demonstrate how our screening rules work.

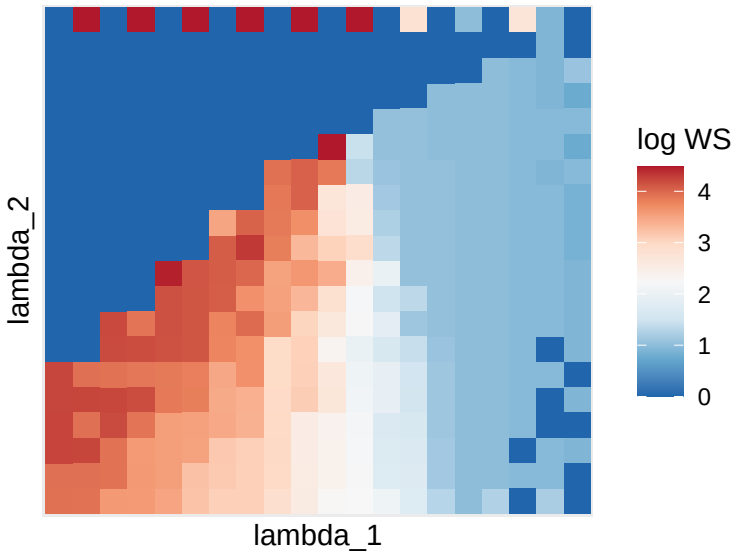
Working sets (WS) are the sets of variables left after we applied our screening rules to the full set of predictors. Histogram of the working set sizes shows that most of the time we have to fit only a handful of variables.

```
hist(fit$working_set_size, breaks = 100, col="blue")
```



2-dimensional plot below shows the log(WS size) for each (λ_1, λ_2) fit.

```
df = data.frame(lambda_1_factor = factor(fit$lambda_1),
                lambda_2_factor = factor(fit$lambda_2),
                ws = fit$working_set_size)

log_0 = function(x){
  return(ifelse(x == 0, 0, log10(x)))
}

ggplot(df, aes(lambda_1_factor, lambda_2_factor, fill=log_0(ws))) +
  scale_fill_distiller(palette = "RdBu") +
  scale_x_discrete("lambda_1", breaks=c(1)) +
  scale_y_discrete("lambda_2", breaks=c(1)) +
  labs(fill='log WS') +
  geom_tile()
```


lambda_2
lambda_1
log WS
4
3
2
1
0