myOpt package to fit linear models

Luca Aiello, Fabio Piacenza

1. Introduction

The goal of myOpt package is to provide users an efficient way to estimate parameters of a linear model through optimization techniques.

2. Gradient descent method

In the following lines, the results and the computation times are compared, using the function of this package for gradient descent method ("linear_gd_optim") and the standard one ("lm" from the "stats" core package).

```
library(myOpt)

## basic example code

set.seed(8675309)

# data simulation for example purposes

n = 1000

x1 = rnorm(n)
x2 = rnorm(n)
X <- cbind(rep(1,n),x1,x2) # predictor matrix

Y = 5.6 + 2.3*x1 + 8.7*x2 + rnorm(n) # response vector

# random initial values for the parameters of the linear model

par <- rnorm(dim(X)[2])

# the function returns a vector containing the values of the estimated parameters

opt_par <- linear_gd_optim(par, X, Y)

# standard lm function for estimating the parameters

lm_par <- lm(Y ~ x1 + x2)$coefficients

opt_par
#> (Intercept)          x1          x2
#>    5.584470    2.286759    8.717051

lm_par
```

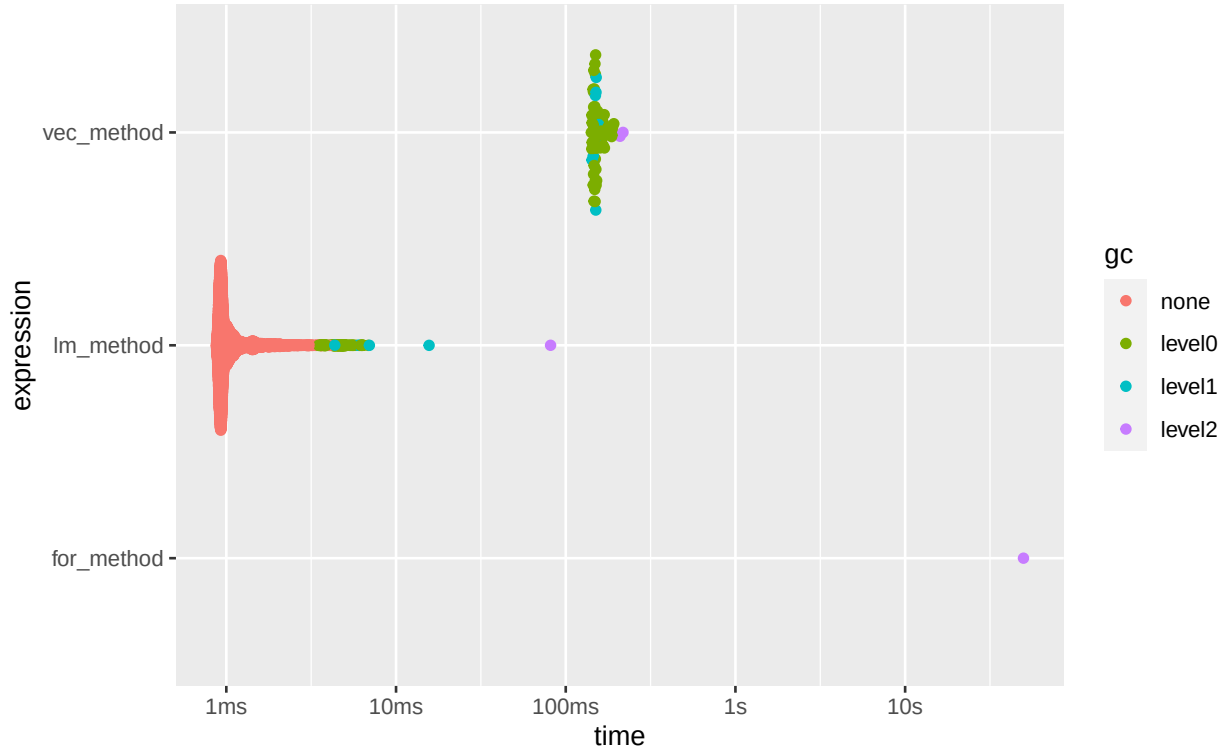
```
#> (Intercept)          x1          x2
#>    5.584780    2.286790    8.717517
```

As it is possible to notice, the results are quite the same, meaning that the accuracy of the method implemented is, in terms of results, comparable to the standard one.

Now let's benchmark the above used vectorized version of the gradient descent method (function "linear_gd_optim") versus the one using for loops (function "linear_gd_optim_old").

```
library(ggplot2)
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
library(tidyr)
library(bench)

## Useful functions

# Plot a benchmark
show_bm <- function(bm) {
  print(print_bench(bm))
  autoplot(bm)
}

# printable bench (for RMarkdown)
print_bench <- function(bm) {
  bm %>%
    mutate(expression = as.character(expression))
}


## Benchmarks

bench::mark(
  lm_method = round(lm(Y ~ x1 + x2)$coefficients,1),
  vec_method = round(linear_gd_optim(par, X, Y),1),
  for_method = round(linear_gd_optim_old(par, X, Y),1),
  filter_gc = FALSE,
  min_time = 10
) %>%
  show_bm()
#> # A tibble: 3 x 13
#>   expression     min  median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
#>   <chr>      <bch:t> <bch:t>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm>
#> 1 lm_method  872.8us 952.8us  887.       220.5KB     13.5  8867   135        10s
#> 2 vec_method 141.6ms 150.4ms    6.38     142.3MB     20.5    65   209      10.2s
#> 3 for_method   49.8s   49.8s    0.0201    55.7KB     23.3     1  1163      49.8s
```

```
#> # ... with 4 more variables: result <list>, memory <list>, time <list>,
#> #   gc <list>
```



From the benchmarks, it is possible to see how the vectorized version of the function performs much faster than the loops-based one; this allows our function to compete with the standard one "lm" not only in terms of result accuracy but also in terms of computation times. In fact, "lm" is still the fastest one (being based on C code internal functions), but "linear_gd_optim" still remains (for this example) within the fractions of a second. On purposes of comparison, these calculations are executed using a processor Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz (4 logical processors) with RAM 24.0 GB, Windows 7 64-bit Operating System, and R version 4.0.2.

3. Steepest descent method

The parameters of the linear model can also be calculated with the steepest descend method, using the function "linear_sd_optim" implemented in this package.

The example of the previous section is repeated applying the steepest descend method, comparing results with the gradient descent method and the standard function "lm".

```
library(myOpt)

## basic example code

set.seed(8675309)

# data simulation for example purposes
```

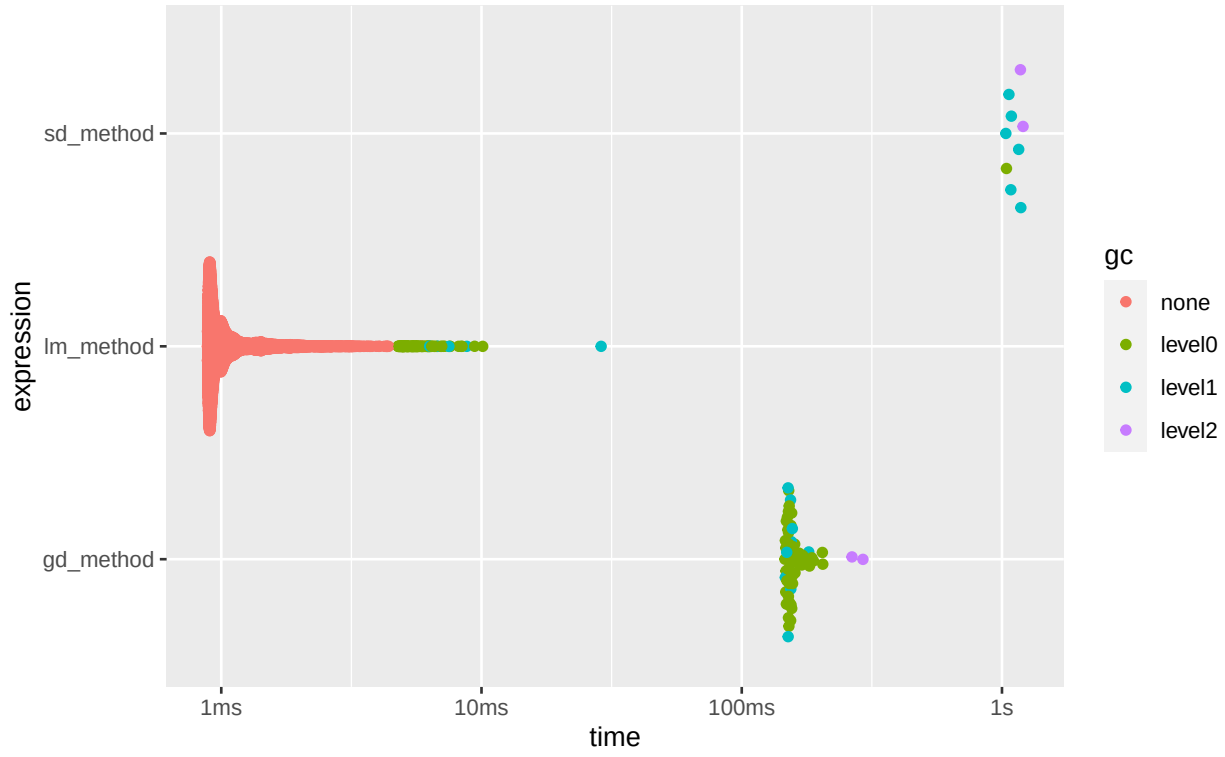
```r
n = 1000

x1 = rnorm(n)
x2 = rnorm(n)
X <- cbind(rep(1,n),x1,x2) # predictor matrix

Y = 5.6 + 2.3*x1 + 8.7*x2 + rnorm(n) # response vector

# random initial values for the parameters of the linear model

par <- rnorm(dim(X)[2])

# the function returns a vector containing the values of the estimated parameters

opt_par <- linear_sd_optim(par, X, Y)

# standard lm function for estimating the parameters

lm_par <- lm(Y ~ x1 + x2)$coefficients

opt_par
#> (Intercept)          x1          x2
#>    5.581628    2.286336    8.712374

lm_par
#> (Intercept)          x1          x2
#>    5.584780    2.286790    8.717517
```

It can be noted that the results are quite similar, meaning that also the accuracy of the implemented steepest descent is, in terms of results, comparable to the standard function "lm".

We proceed benchmarking the functions "linear_gd_optim" (based on the gradient descent) and "linear_sd_optim" (based on the steepest descent) with "lm" (the standard one).

```r
library(ggplot2)
library(dplyr)
library(tidyr)
library(bench)

## Useful functions

# Plot a benchmark
show_bm <- function(bm) {
  print(print_bench(bm))
  autoplot(bm)
}

# printable bench (for RMarkdown)
print_bench <- function(bm) {
  bm %>%
    mutate(expression = as.character(expression))
}
```

```r
## Benchmarks

bench::mark(
  lm_method = round(lm(Y ~ x1 + x2)$coefficients,1),
  gd_method = round(linear_gd_optim(par, X, Y),1),
  sd_method = round(linear_sd_optim(par, X, Y),1),
  filter_gc = FALSE,
  min_time = 10
) %>%
  show_bm()
#> # A tibble: 3 x 13
#>   expression      min   median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc
#>   <chr>      <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl> <int> <dbl>
#> 1 lm_method  884.39us  966.5us    877.        220KB     9.40  8769    94
#> 2 gd_method  146.58ms 154.07ms      6.18      142MB    23.1     62   232
#> 3 sd_method     1.03s    1.09s      0.897     803MB    18.3      9   184
#> # ... with 5 more variables: total_time <bch:tm>, result <list>, memory <list>,
#> #   time <list>, gc <list>
```



From the benchmarks, it is possible to observe that the gradient descent method is faster than the steepest descent one, in front of a comparable accuracy. This is essentially due to the additional operations that the steepest descent method has to perform at each iteration of the optimization procedure to calculate the step, compared to the gradient descent one.

4. Gradient and steepest descent methods using more predictors

This section shows that the steepest gradient and the steepest descent methods, implemented in this package, work using even more than two predictors.

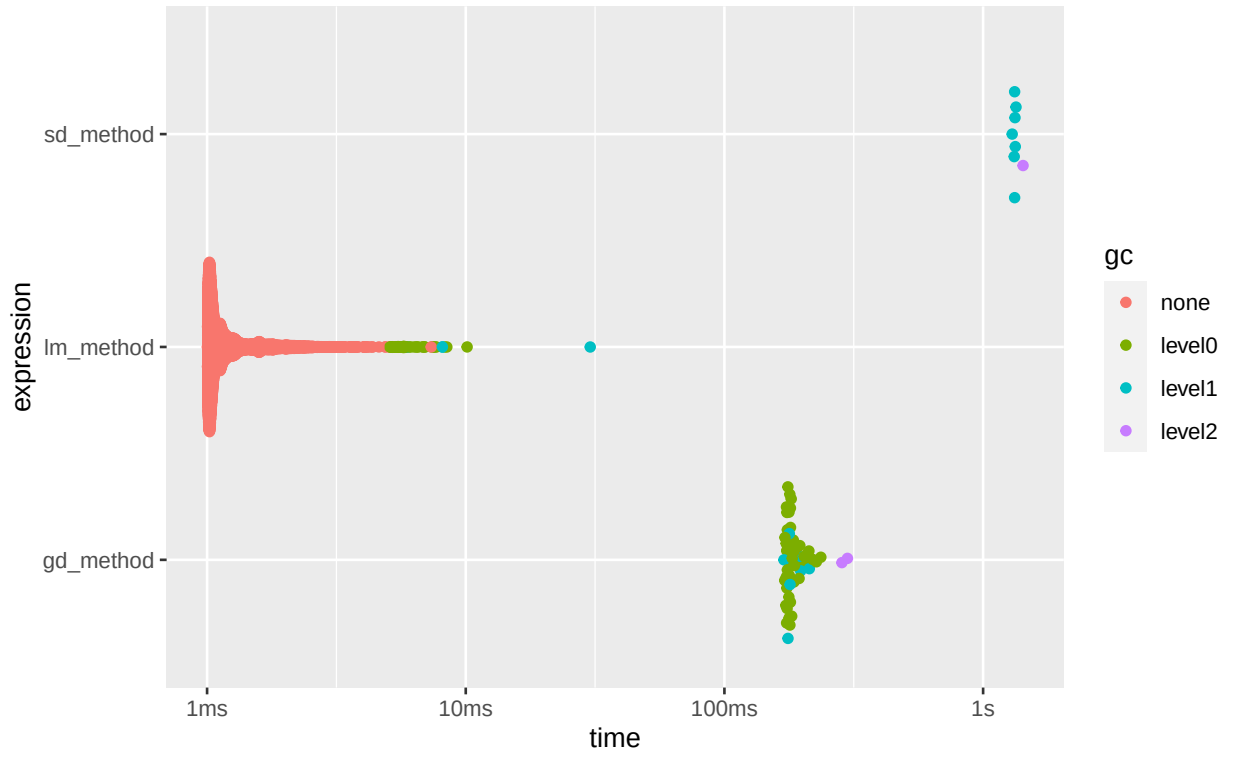
A new example, including three predictors, is reported in this section section applying the gradient descend and the steepest descend methods, comparing results with the standard function "lm".

```
library(myOpt)

## basic example code

set.seed(8675309)

# data simulation for example purposes

n = 1000

x1 = rnorm(n)
x2 = rnorm(n)
x3 = rnorm(n)
X <- cbind(rep(1,n),x1,x2,x3) # predictor matrix

Y = 5.6 + 2.3*x1 + 8.7*x2 + 7.2*x3 + rnorm(n) # response vector

# random initial values for the parameters of the linear model

par <- rnorm(dim(X)[2])

# the function returns a vector containing the values of the estimated parameters

opt_par_gd <- linear_gd_optim(par, X, Y)
opt_par_sd <- linear_sd_optim(par, X, Y)

# standard lm function for estimating the parameters

lm_par <- lm(Y ~ x1 + x2 + x3)$coefficients

opt_par_gd
#> (Intercept)          x1          x2          x3
#>    5.583167    2.319347    8.654901    7.199944

opt_par_sd
#> (Intercept)          x1          x2          x3
#>    5.578298    2.318367    8.650892    7.196371

lm_par
#> (Intercept)          x1          x2          x3
#>    5.583666    2.319455    8.655211    7.200265
```

It can be noted that the results are quite similar, meaning that the accuracy of the implemented steepest gradient and steepest descent methods is, in terms of results, comparable to the standard function "lm", also using three predictors.

We proceed benchmarking the functions "linear_gd_optim" (based on the gradient descent) and "linear_sd_optim" (based on the steepest descent) with "lm" (the standard one) using three predictors.

```
library(ggplot2)
library(dplyr)
library(tidyr)

## Useful functions

# Plot a benchmark
show_bm <- function(bm) {
  print(print_bench(bm))
  autoplot(bm)
}

# printable bench (for RMarkdown)
print_bench <- function(bm) {
  bm %>%
    mutate(expression = as.character(expression))
}


## Benchmarks

bench::mark(
  lm_method = round(lm(Y ~ x1 + x2 + x3)$coefficients,1),
  gd_method = round(linear_gd_optim(par, X, Y),1),
  sd_method = round(linear_sd_optim(par, X, Y),1),
  filter_gc = FALSE,
  min_time = 10
) %>%
  show_bm()
#> # A tibble: 3 x 13
#>   expression      min   median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc
#>   <chr>      <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl> <int> <dbl>
#> 1 lm_method       1ms   1.08ms   782.     267.65KB     7.30  7820    73
#> 2 gd_method   169.9ms 179.93ms     5.29   180.91MB    20.5     53   205
#> 3 sd_method      1.3s    1.33s     0.748    1.01GB    15.3      8   164
#> # ... with 5 more variables: total_time <bch:tm>, result <list>, memory <list>,
#> #   time <list>, gc <list>
```



The benchmarks confirm that the gradient descent method is faster than the steepest descent one, in front of a comparable accuracy, also using three predictors, and that "lm" remains the fastest one.

5. Prediction

After the estimation of the parameters, for both method it can be applied a prediction function, that predicts the outcome given the estimated parameters vector of the model and a predictors data matrix.

```
# prediction with the parameters estimated through the lm function
prediction_lm <- predict(lm(Y ~ x1 + x2 + x3))

# prediction with the parameters estimated through the gradient descend method
prediction_gd <- my_linear_predict(opt_par_gd, X)

# prediction with the parameters estimated through the steepest descend method
prediction_sd <- my_linear_predict(opt_par_sd, X)
```

After the computation of the predictions this package allows also to compute the error associated with the prediction.

```
# prediction error associated to the lm function estimated parameters
my_pred_error(Y, prediction_lm)
#> [1] 0.9872884

# prediction error associated to the gradient descend estimated parameters
my_pred_error(Y, prediction_gd)
#> [1] 0.9872888
```

```
# prediction error associated to the steepest descend estimated parameters
my_pred_error(Y, prediction_sd)
#> [1] 0.987355
```

It is possible to see how the three methods give similar results in terms of prediction and also in terms of prediction error.

6. Cross validation

In order to estimate how accurately the predictive model will perform in practice, cross validation techniques can be used. In particular, the k-fold cross validation is implemented in the myOpt package to assess the predictions of the gradient and steepest descent methods. The function "my_k_fold_cv" can be used on this purpose for both methods, depending on the value of the argument "method" (to be set to "gd" for gradient descent, and to "sd" for steepest descent).

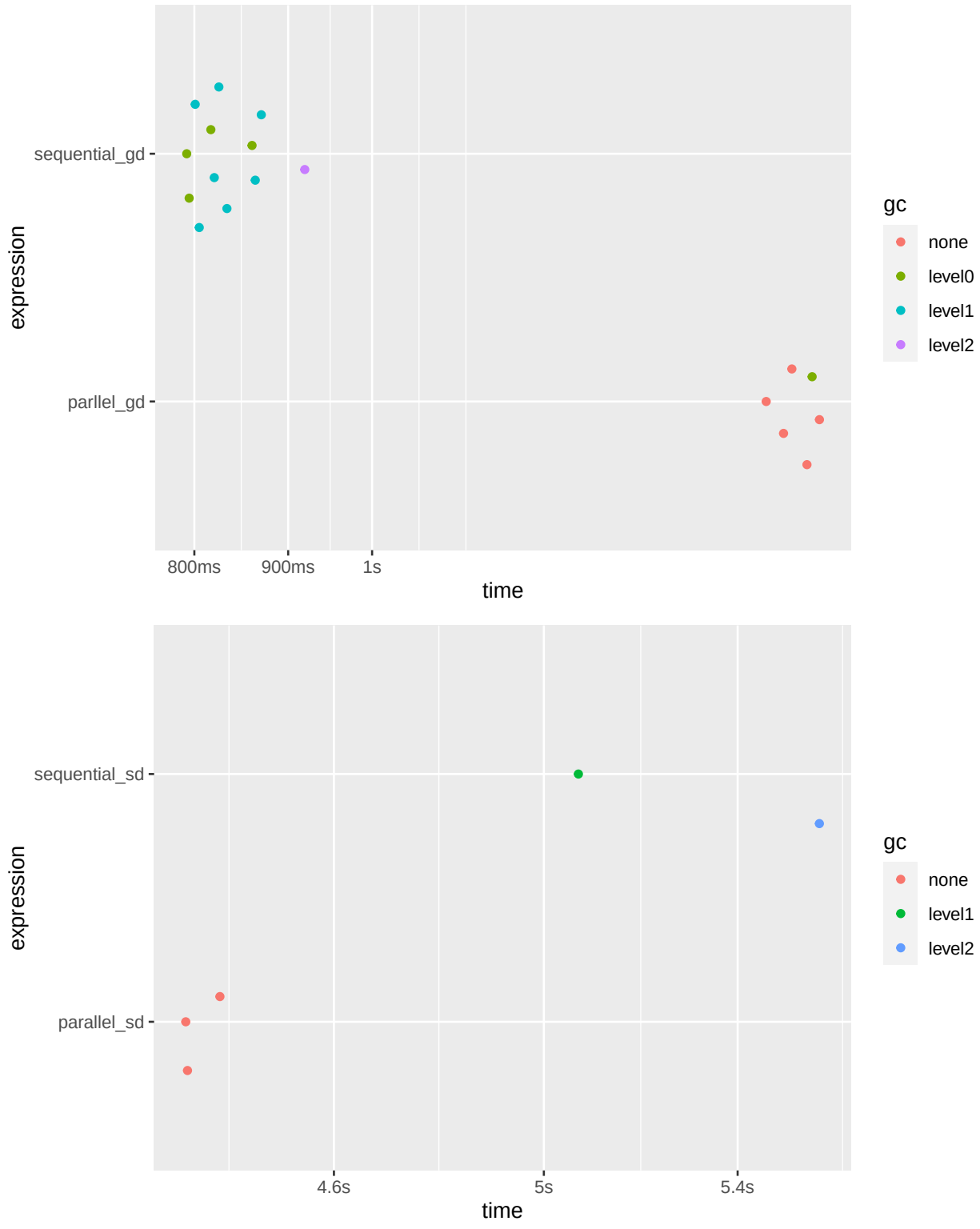
```
library(doSNOW)
#> Warning: package 'doSNOW' was built under R version 4.0.4
#> Loading required package: foreach
#> Loading required package: iterators
#> Loading required package: snow
#> Warning: package 'snow' was built under R version 4.0.4

K <- 5
set.seed(8675309)
my_k_fold_cv(par, X, Y, K)
#> [1] 0.9992729
set.seed(8675309)
my_k_fold_cv(par, X, Y, K, method = "sd")
#> [1] 0.9992957
```

We can observe that the two methods have very similar prediction errors, which are, as expected, a bit higher than the in-sample errors calculated in the previous section.

The function "my_k_fold_cv" can also perform a parallel calculation. It is sufficient to set the argument "parallel" equal to TRUE to move from the sequential to the parallel computing, distributing the calculation on all the available cores (which are 4 in this example). The impact of the parallel computing on the calculation times is analyzed below.

```
## Useful functions

# Plot a benchmark
show_bm <- function(bm) {
  print(print_bench(bm))
  autoplot(bm)
}

# printable bench (for RMarkdown)
print_bench <- function(bm) {
  bm %>%
    mutate(expression = as.character(expression))
}
```

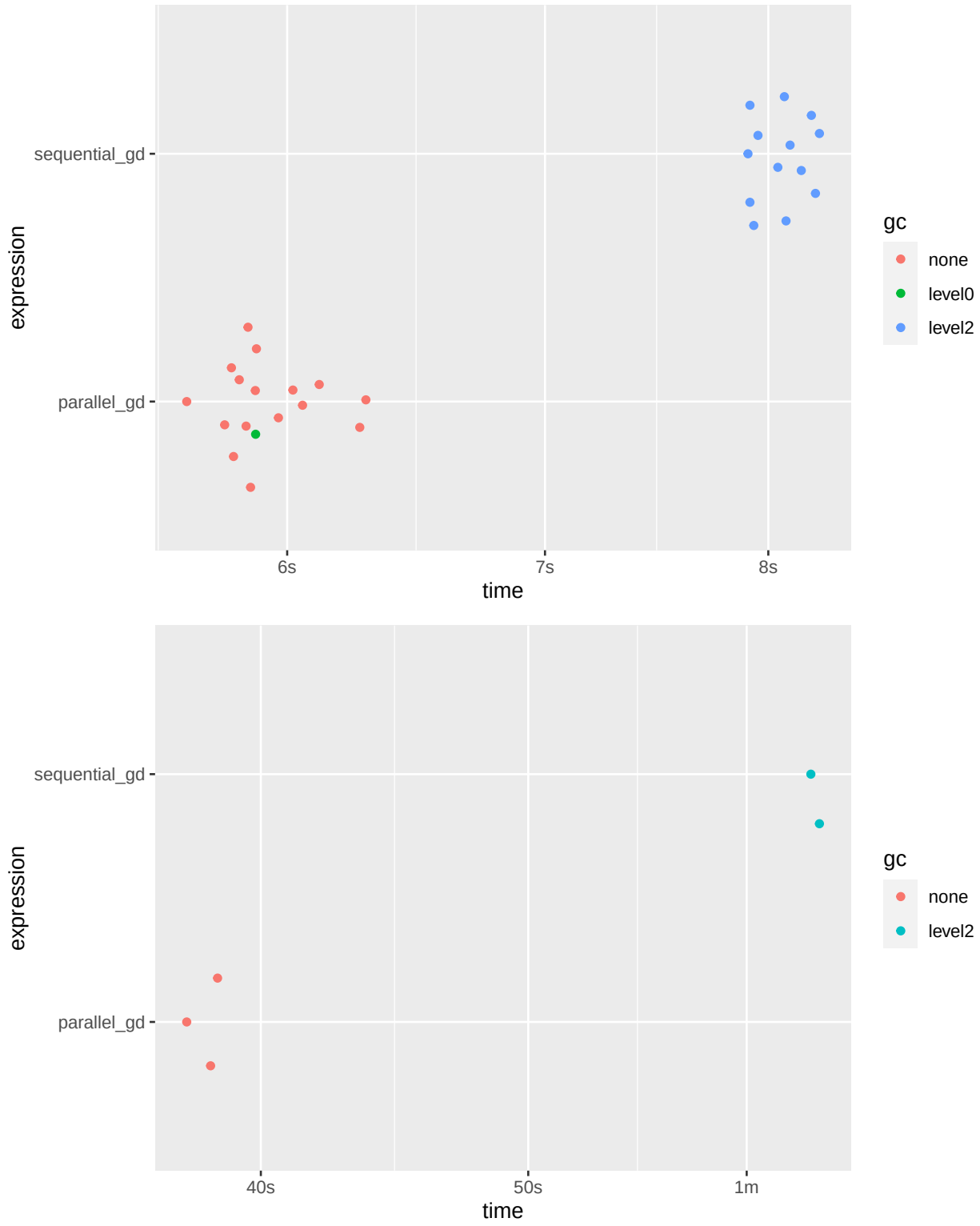
```r
set.seed(8675309)
bench::mark(
  sequential_gd = round(my_k_fold_cv(par, X, Y, K),1),
  parllel_gd = round(my_k_fold_cv(par, X, Y, K, parallel = TRUE),1),
  filter_gc = FALSE,
  min_time = 10
) %>%
  show_bm()
#> # A tibble: 2 x 13
#>   expression         min   median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc
#>   <chr>         <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl> <int> <dbl>
#> 1 sequential_gd 792.37ms 822.64ms     1.20   725.39MB  17.7       12   177
#> 2 parllel_gd       1.64s    1.71s     0.587    3.29MB   0.0978     6     1
#> # ... with 5 more variables: total_time <bch:tm>, result <list>, memory <list>,
#> #   time <list>, gc <list>

set.seed(8675309)
bench::mark(
  sequential_sd = round(my_k_fold_cv(par, X, Y, K, method = "sd"),1),
  parallel_sd = round(my_k_fold_cv(par, X, Y, K, method = "sd", parallel = TRUE),1),
  filter_gc = FALSE,
  min_time = 10
) %>%
  show_bm()
#> # A tibble: 2 x 13
#>   expression      min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
#>   <chr>        <bch:> <bch:>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm>
#> 1 sequential_~  5.07s  5.32s     0.188    3.74GB     11.6     2   124      10.6s
#> 2 parallel_sd   4.34s  4.34s     0.229    2.83MB      0       3     0      13.1s
#> # ... with 4 more variables: result <list>, memory <list>, time <list>,
#> #   gc <list>
```



It can be observed that, considering the current example, the parallel computing is convenient for steepest descent method, whereas it increases the computation time for the gradient descent one. For this last case, since also distributing the calculation has a time impact, it can happen that it more than compensates the time saving obtained splitting the procedure on more cores.

In order to fully appreciate the benefit of the parallel computing, we can set a more computational intensive example, increasing the sample size from 1000 to 10000, and the number of folds from 5 to 10.

```
set.seed(8675309)
n <- 10000
x1 <- rnorm(n)
x2 <- rnorm(n)
Y <- 1 + 0.5*x1 + 0.2*x2 + rnorm(n)
X <- cbind(rep(1,n),x1,x2)
par <- rnorm(dim(X)[2])
K <- 10

set.seed(8675309)
bench::mark(
  sequential_gd = round(my_k_fold_cv(par, X, Y, K),1),
  parallel_gd = round(my_k_fold_cv(par, X, Y, K, parallel = TRUE),1),
  filter_gc = FALSE,
  min_time = 100
) %>%
  show_bm()
#> # A tibble: 2 x 13
#>   expression      min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
#>   <chr>        <bch:> <bch:>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm>
#> 1 sequential_~   7.9s  8.08s     0.124   10.19GB 20.0        13  2094      1.75m
#> 2 parallel_gd   5.65s  5.89s     0.169    4.33MB  0.00992    17     1      1.68m
#> # ... with 4 more variables: result <list>, memory <list>, time <list>,
#> #   gc <list>

set.seed(8675309)
bench::mark(
  sequential_gd = round(my_k_fold_cv(par, X, Y, K, method = "sd"),1),
  parallel_gd = round(my_k_fold_cv(par, X, Y, K, method = "sd", parallel = TRUE),1),
  filter_gc = FALSE,
  min_time = 100
) %>%
  show_bm()
#> # A tibble: 2 x 13
#>   expression      min median `itr/sec` mem_alloc `gc/sec` n_itr  n_gc total_time
#>   <chr>       <bch:t> <bch:>     <dbl> <bch:byt>    <dbl> <int> <dbl>   <bch:tm>
#> 1 sequential~   1.05m  1.06m    0.0157   70.44GB     9.64     2  1225      2.12m
#> 2 parallel_gd   37.6s 38.36s    0.0262    4.33MB     0        3     0      1.91m
#> # ... with 4 more variables: result <list>, memory <list>, time <list>,
#> #   gc <list>
```



Considering this more computational intensive example, it can be noted that the parallel computing allows to save time also using the gradient descent method.

7. Conclusions

This document shows how to apply the functions to estimate the linear regression parameters, implemented in this package, based on the gradient descent and the steepest descent methods.

Several examples are reported, including comparisons and benchmarks between the two implemented methods, and versus the standard function "lm". All the provided results agree on the fact that the implemented methods are comparable, in terms of accuracy, with respect to the standard function.

In terms of computation time, gradient descent method is faster than the steepest descent one, because of the simpler step calculation at each iteration. It can observed that, even if it is still slower, the gradient descent method reaches computation times which are comparable to the standard function, keeping (at least, on considered examples) an order of fractions of seconds.

The prediction of the gradient descent and the steepest descent methods is evaluated, showing that their in-sample prediction error is comparable to the one of the standard function.

Some examples of k-fold cross validation are reported, showing how to optimize the calculations using the parallel computing. These examples show that the implemented gradient descent and steepest descent methods have also a comparable out-of-sample prediction error.