PARALLEL COMPUTING LAB REPORT

AN IMPLEMENTATION OF SELECTION PROCEDURE FOR THE LOGIT MODEL

Course: Parallel Computing

Florentin Coeurdoux

10. 01. 2020

Referee: Prof. Mathieu Marbac

Contents

Contents

1 Introduction

Our aim for this lab is to build a procedure capable of selecting variables in the logit model. To perform this goal we will do a stepwise search to optimize the prediction error estimated by cross-validation.
First I will introduce the statistical model which I will use, then I will explain how I programmed the variable selections for the model optimizing the prediction error. Afterwards, I will present a few exceptions which I have managed, I will demonstrate how I have conducted code profiling, and I will show how I have considered parallel computing for this procedure. Lastly, I will illustrate my procedure with a numerical experiment.

2 The Logit model

Let $(X_1^T, Y_1), ..., (X_n^T, Y_n)$ be observed independent copies of the random vector (X, Y) with $X \in \mathbb{R}^d$ and $Y \in \{0, 1\}$.
The distribution of Y given $X = x$ is assumed to be a logit model such that :
$\mathbb{P}(Y = 1|X = x) = \frac{exp(x^T\beta)}{1+exp(x^T\beta)}$ and $\mathbb{P}(Y = 0|X = x) = 1 - \mathbb{P}(Y = 1|X = x)$
where $\beta \in \mathbb{R}^p$ is the vector of the model parameters.

3 Programming the variables selection

3.1 The *rlogit* function

First we had to implement the function rlogit which generate observations from a logit model. I created the following function in the script entitled Data_generation.R :

```
rlogit <- function(n, beta){
  X <- matrix(rnorm(n*length(beta),mean=0,sd=1), n, length(beta))
  U <- (exp(X %*% t(beta))) / (1 +exp(X %*% t(beta)))
  Y <- (runif(n) < U)*1
```

```
  out <- data.frame(Y, X)
  return(out)
}
```

This function take as imput parameter n the number of observation, and *beta* the set of coefficient of the logit regression : $\frac{exp(x^T\beta)}{1+exp(x^T\beta)}$. X is made randomly according to a gaussian law, *beta* is used to compute the probabilities U, and then we design y according to the probabilities U. At the output we obtain a dataframe compose of the n variables X and the response variable Y.

3.2 The *rhapson_newton* function

In order to program the *rhapson_newton* function, I had to program the log likelihood, the gradient and the hessian functions. All these functions are stored in the script entitled : basic_mle.R

- The *loglikl* function :

The following function computes the log likelihood of the model :

```
loglikl <- function(x,y,beta) {
  if(length(beta) != ncol(x)){
    return("The number of values in beta should be equal to the number of
           variable of X")}
  else{
  if(length(beta)==1){
    u = exp(x * beta) / (1 + exp(x * beta))
    l=0
    for (i in 1:length(y)) {
      l = l - (y[i]*log(u[i]) + (1-y[i])*log(1-u[i]))
    }}
  else{
    u = exp(x %*% beta) / (1 + exp(x %*% beta))
    l=0
    for (i in 1:nrow(x)) {
```

```
      l = l - (y[i]*log(u[i]) + (1-y[i])*log(1-u[i]))
    }}
  return(l) }
}
```

The function takes as input three arguments : the variables x, the observations y, and a value for beta. It returns the value of the log likelihood.

- The *gradient* function :

The following function computes the gradient of the model. The latter can be written as follows : $\frac{\partial \ell}{\partial \beta_i} = \sum x_i(y_i - \frac{exp(x_i^T\beta)}{1+exp(x_i^T\beta)})$

```
gradient = function(x,y,beta) {
  if(length(beta) != ncol(x)){
    return("The number of values in beta should be equal to the number of
           variable of X")}
  else{
  if(length(beta)==1){
    u = exp(x * beta) / (1 + exp(x * beta))
    for (i in 1:length(test)) {g = g + test[i]*(u[i]-y[i])}
  }else{
    g= rep(0,ncol(x))
    u = exp(x %*% beta) / (1 + exp(x %*% beta))
    for (i in 1:nrow(x)) {g = g + x[i,]*(u[i]-y[i])}
  }
  return(g) }
}
```

The function takes as input three arguments : the variables x, the observations y, and a value for *beta*. It returns a vector of length n containing the gradient.

- The *hessian* function :

The script computes the hessian of the model. The latter can be written as follows : $\frac{\partial \ell^2}{\partial \beta_i \beta_h} = \sum (\frac{exp(x_i^T\beta)}{1+exp(x_i^T\beta)})(1 - \frac{exp(x_i^T\beta)}{1+exp(x_i^T\beta)})x_i x_i^T$

```
hessian = function(x,y,beta) {
  if(length(beta) != ncol(x)){
    return("The number of values in beta should be equal to the number of
           variable of X")}
  else{
  if(length(beta)==1){
    u = exp(x * beta) / (1 + exp(x * beta))
    h = u[i]*(1-u[i]) * t(x) %*% x}
  else{
    u = exp(x %*% beta) / (1 + exp(x %*% beta))
    h = matrix(0,ncol(x),ncol(x))
    for (i in 1:nrow(x)) {
      h = h + u[i]*(1-u[i]) * t(x) %*% x}
  }
  return(h)}
}
```

The function takes as input three arguments : the variables x, the observations y, and a value for *beta*. It returns a matrix of length $n * n$ containing the hessian.

- The *rhapson_newton* function :

The script performs a Newton-Raphson algorithm to approximate a value of $\hat{\beta}$. This algorithm approximates successively by the application of a formula to a point, the point in which the function is maximized. The hessian being inversible, twice derivable, and definite negative, therefore ℓ is concave and admits a unique minimum.

We initialize the algorithm with a θ chosen in $\mathbb{R}$. Here we chose values close to the chosen β. We then implement the algorithm with the following formula :

$\theta_{k+1} = \theta_k - \underbrace{(\nabla^2 \ell(\theta_k))^{-1}}_{Hessian} \underbrace{\nabla \ell(\theta_k)}_{Gradient}$

We iterate the algorithm until reaching a small enough margin of error, i.e. $|\theta_{k+1} - \theta_k| <$ *margin error*. For this purpose, we create a function *norme*, returning the root of the sum of the squared values of a vector.

With this algorithm, we finally find the approximate value of $\hat{\beta}$ that we are looking for.

```
rhapson_newton <- function(x,y,beta0,eps) {
  if(length(beta0) != ncol(x)){return("The number of values in beta should be
                                      equal to the number of variable of X")} else {
  beta = beta0
  if(length(beta)==1){
    dir = eps+1
    while(sqrt(dir)^2>eps)  {
      grd= gradient(x,y,beta)
      hes= hessian(x,y,beta)
      dir=(1/hes)*grd
      beta = beta - dir
    }}
  else{
    dir = rep(eps+1,length(beta))
    while(norm_vec(dir)>eps)  {
      grd= gradient(x,y,beta)
      hes= hessian(x,y,beta)
      dir = solve(hes) %*% grd
      beta = beta - dir
    }}
  return(beta) }
}
```

The function takes as input four arguments : the variables x, the observations y, a value of an aproximation of *beta* and a value for the error. It returns a vector containing an estimation of the true *beta*.

3.3 The *basic.cv* function :

Here our aim is to implement a function which returns the estimator of the error of prediction obtained by cross-validation for any subset of covariates by using the MLE of the model. This script computes the cross-validation error of the model with the "leave one out method".

```
basiv.cv <- function(x, y, beta0) {
  pred_error_sq <- rep(0, length(y))
  for(i in 1:length(y)) {
    x_i <- x[-i, ] # leave i'th observation out
    y_i <- y[-i]
    mdl <- rhapson_newton(x_i, y_i, beta0,0.1)
    u_pred <- (exp(x %*% mdl)) / (1 +exp(x %*% mdl))
    y_pred <- (runif(length(y)) < u_pred)*1
    pred_error_sq[i] <-sum((y_pred - y)^2) # sum squared prediction errors
  }
  error_model <- mean(pred_error_sq) # the mean of the error as the indicator
  return(error_model)
}
```

To do so I remove each of the observations one time, compute a new model from the data subset and compute the mean of the error. This function is stored in the script entitled : cross_validation_loo.R

3.4 The *basic.modelcomparison* function :

Here our aim is to implement a function that takes as input : the sample and a set of competing models. This returns the best model and the prediction error thancks to a choosen criteria. I choose to compare the models thancks to three different indicators : R-squared, AIC and the previous cross validation error :

Choose the best model according to the R-squared criteria

```
basic.modelcomparison_Rsqared <- function(x, y, set.of.beta) {
  mt_R_squared <- matrix(0, 1, dim(set.of.beta)[1])
  total_var <- sum((y-mean(y))^2)
  for(i in 1:dim(set.of.beta)[1]) {
    u_pred <- (exp(x %*% set.of.beta[i,])) / (1 +exp(x %*% set.of.beta[i,]))
    y_pred <- (runif(length(y)) < u_pred)*1
    explained_var <- sum((y_pred - mean(y))^2)
```

```
    mt_R_squared[i] <- 1 - (explained_var/total_var)
  }
  nb_model <- which.max(mt_R_squared)
  return(list(R_squared_best_model = mt_R_squared[nb_model],
              best_model = set.of.beta[nb_model,]))
}
```

Choose the best model according to the AIC criteria

```
basic.modelcomparison_AIC <- function(x, y, set.of.beta) {
  mt_AIC <- matrix(0, 1, dim(set.of.beta)[1])
  K <-dim(set.of.beta)[2]
  for(i in 1:dim(set.of.beta)[1]) {
    mt_AIC[i] <- -2*loglikl(x, y, set.of.beta[i,]) + 2*K
    + (2*K*(K+1)/(length(y)-K-1)) #Corrected AIC for K small <40
  }
  nb_model <- which.min(mt_AIC)
  return(list(AIC_best_model = mt_AIC[nb_model],
              best_model = set.of.beta[nb_model,]))
}
```

Choose the best model according to the cross validation error

```
basic.modelcomparison_CV <- function(x, y, set.of.beta) {
  mt_CV <- matrix(0, 1, dim(set.of.beta)[1])
  for(i in 1:dim(set.of.beta)[1]) {
    mt_CV[i] <- basiv.cv(x,y, beta0)
  }
  nb_model <- which.min(mt_CV)
  return(list(CV_best_model = mt_CV[nb_model],
              best_model = set.of.beta[nb_model,]))
}
```

Choose the best model according to a subset of the relevant variables, and a choosen criteria.

```
basic.modelcomparison <- function(x, y, models, beta0, selection = "all") {
  if(length(beta0) != ncol(x) | ncol(models) != ncol(x)){
    return("The number of values in beta should be equal to the number of
           variable of X")}
  else {
  bet <- matrix(0, ncol = ncol(x), nrow=dim(models)[1])
  for(i in 1:dim(models)[1]){
    combi = models[i,]
    betaa= rhapson_newton(x, y, beta0*combi, 0.1)
    bet[i,] = combi
  }
  if(selection == "AIC"){basic.modelcomparison_AIC(x, y, bet)}
  else if(selection == "Rsqared"){basic.modelcomparison_Rsqared(x, y, bet)}
  else if(selection == "cv"){basic.modelcomparison_CV(x, y, bet)}
  else if(selection == "all"){list(
    AIC_model = basic.modelcomparison_AIC(x, y, bet),
    Rsqared_model = basic.modelcomparison_Rsqared(x, y, bet),
    CV_model = basic.modelcomparison_CV(x, y, bet))}
  else{
    return("You have to choose a selection method between CV,
           AIC or R-squared")} }
}
```

3.5 The *basic.modelselection* function :

Here our aim is to implement a function that takes as input argument the sample and the direction (backward or forward).This returns the best model and its estimator of the prediction error.

```
basic.modelselection <- function(x, y, direction = "forward", beta0, selection = "all"){
  if(length(beta0) != ncol(x)){
    return("The number of values in beta should be equal to the number
           of variable of X")} else {
```

```
if(tolower(direction)=="forward"){
  # Select the best model for all 1 variables models (Identity matrix)
  models <- diag(dim(x)[2])
  best_model <- basic.modelcomparison(x, y, models, beta0, selection)$best_model
  # Select the best model between combinations of p variables
  #and the previously selected
  for(p in 2:ncol(x)){
    models <- matrix(0, ncol=ncol(x), nrow=choose(ncol(x),p))
    while( all(rowMeans(models)==rep( mean(c(rep(1,p),rep(0,ncol(x)-p))),ncol(x)))==FALSE
      for(i in 1:nrow(models)){
        combin <- sample(1:ncol(x),p)
        row <- rep(0,ncol(x))
        for(j in combin){row[j]<-1}
        ajouter<- TRUE
        for(k in 1:nrow(models)){ if( all(row==models[k,]) ){
          ajouter <- FALSE
          break}}
        if(ajouter==TRUE){
          for(m in 1:nrow(models)){if( all(models[m,]==rep(0,ncol(X))) ){
          models[m,]<- row
          break}} }
      } }
    models = rbind(models, unlist(best_model, use.names=FALSE))
    best_model <- basic.modelcomparison(x, y, models, beta0, selection)
    return(best_model)
  } }
else if(tolower(direction)=="backward"){
  # Select the best model for all variables models
  models <- matrix(1, ncol=ncol(x), nrow=1)
  best_model <- basic.modelcomparison(x, y, models, beta0, selection)$best_model
  # Select the best model between combinations of p variables
  #and the previously selected best model
  for(p in ncol(x):1){
```

```
      models <- matrix(0, ncol=ncol(x), nrow=choose(ncol(x),p))
      while( all(rowMeans(models)==rep( mean(c(rep(1,p),rep(0,ncol(x)-p))),ncol(X)))==FALSE
        for(i in 1:nrow(models)){
          combin <- sample(1:ncol(x),p)
          row <- rep(0,ncol(x))
          for(j in combin){row[j]<-1}
          ajouter<- TRUE
          for(k in 1:nrow(models)){ if( all(row==models[k,]) ){
            ajouter <- FALSE
            break}}
          if(ajouter==TRUE){
            for(m in 1:nrow(models)){if( all(models[m,]==rep(0,ncol(x))) ){
            models[m,]<- row
            break}} }
        } }
      models = rbind(models, unlist(best_model, use.names=FALSE))
      best_model <- basic.modelcomparison(x, y, models, beta0, selection)
      return(best_model)
    }
  }
  else {return("Please enter backward or forward as direction in the function ")} }
}
```

This function allows to choose the direction of the steop wise selection and the criterium of the selection.

4 Managing exeptions

In order to prevent my scrpit from issues, and to perform well when using different data set, I do considered some exeptions.

- Observations x compose of one variable, one column :

To manage this case, I changed the four function : *loglik*, *gradient*, *hessian* and *rhapson_newton* to be able to perform the calculation.

- If *beta* and x can't match :

Here I choose to tell the user that there is an issue about the size of x or beta. for exemple here :

```
gradient(x, y, beta[-1])
```

```
#  [1] "The number of values in beta should be equal to the number of\n          variable c
```

- If the selection method for the *basic.modelcomparison* and *basic.modelselection* functions is not well writen :

I convert the entry in lower case in case of mistake with capital letter, and if the choosen criteria did not match with the ones which exists then I inform the user of the different possibilities :

```
basic.modelselection(x, y, "baCKward", beta0)$CV_model
```

```
#  $CV_best_model
#  [1] 5.8
#
#  $best_model
#  [1] 1 1 1 1 1
```

```
basic.modelcomparison(x, y, models, beta0, selection = "")
```

```
#  [1] "You have to choose a selection method between CV, \n          AIC or R-squared"
```

- If the direction for the *basic.modelselection* function is not well writen :

I convert the entry in lower case in case of mistake with capital letter, and if the choosen direction did not match with the ones which exists then I inform the user of the different possibilities :

```
basic.modelselection(x, y, "test", beta0)
```

```
#  [1] "Please enter backward or forward as direction in the function "
```

5 Code profiling

By doing the code profiling, we observe that the most time consuming part is the one from the script basic_mle.R, more precisely the function *hessian*, *gradient* and the command *solve* in the *rhapson_newton* function. So these gunctions are the ones we must optimised. We also observed that the *basic.modelcomparison* is very fast hwen using the AIC criteria compared to using the cross validation, this is due to the fact the the cross validation uses the *rhapson_newton* function multiple times, while AIC only uses ty *loglik* function.
We also observe that within these time and memory consuming functions, the most costly part is the multiplication of matrices. Unfortunately this computation can't be vectorized. There is three ways to optimase large matrix multiplication in R : using the function crossprod(A, B) instead of t(A)%*%B, use Rccp and code the function in C++ or use GPU computing with package such as gmatrix. Unfortunatily I don't have a GPU and I don't know the language C++. so I decided to replace the matrix multiplication factor by the crossprod function.

Exemple here with the gradient and hessian functions :

```
gradient_new <-  function(x,y,beta) {
  if(length(beta) != ncol(x)){return("The number of values in beta should
                                     be equal to the number of variable of X")} else{
  if(length(beta)==1){
    u = exp(x * beta) / (1 + exp(x * beta))
    for (i in 1:length(test)) {g = g + test[i]*(u[i]-y[i])}
  }else{
    g= rep(0,ncol(x))
    u = exp(crossprod(x,beta)) / (1 + exp(crossprod(t(x),beta)))
    for (i in 1:nrow(x)) {g = g + x[i,]*(u[i]-y[i])}
  }
  return(g) }
}

hessian_new <-  function(x,y,beta) {
  if(length(beta) != ncol(x)){
    return("The number of values in beta should be equal to the number of
           variable of X")}
```

```
  else{
    if(length(beta)==1){
      u = exp(x * beta) / (1 + exp(x * beta))
      h = u[i]*(1-u[i]) * t(x) %*% x}
    else{
      u = exp(crossprod(t(x),beta))  / (1 + exp(crossprod(t(x),beta)) )
      h = matrix(0,ncol(x),ncol(x))
      for (i in 1:nrow(x)) {
        h = h + u[i]*(1-u[i]) * crossprod(x,x)}
    }
    return(h)}
}
```

After replacing all the matrix multiplication factors by the crossprod function we observed a not negligible improvement, from 33450ms without crossprod we did the cross validation in only 30590ms for the same task and the same 1000 observations. (the corresponding optimized profiling file is opimized_cd.Rproviz, and the basic one is cross_validation_profile.Rproviz)

Know the second possible optimization is vectorizing the different for loops in the *gradient* and *hessian* function. I optimize these for loops by using the sapply function. Here is an exemple with the gradient and the hessian :

```
gradient_new <-  function(x,y,beta) {
  if(length(beta) != ncol(x)){return("The number of values in beta should
    be equal to the number of variable of X")} else{
      if(length(beta)==1){
        u = exp(x * beta) / (1 + exp(x * beta))
        for (i in 1:length(test)) {g = g + test[i]*(u[i]-y[i])}
      }else{
        g= rep(0,ncol(x))
        u = exp(crossprod(t(x),beta)) / (1 + exp(crossprod(t(x),beta)))
        vec = sapply(1:nrow(x),function(i){x[i,]*(u[i]-y[i])})
      }
      return(as.matrix(rowSums(vec))) }
```

```
}


hessian_new <-  function(x,y,beta) {
  if(length(beta) != ncol(x)){return("The number of values in beta should
    be equal to the number of variable of X")} else{
    if(length(beta)==1){
      u = exp(crossprod(x,beta))  / (1 + exp(crossprod(x,beta)) )
      h = u[i]*(1-u[i]) * crossprod(x, x)}
    else{
      u =exp(crossprod(t(x),beta))  / (1 + exp(crossprod(t(x),beta)) )
      vec = sapply(1:nrow(x),function(i){(u[i]*(1-u[i]))*crossprod(x, x)})
      vec =matrix(rowSums(vec), ncol=ncol(x), nrow = ncol(x))
      return(vec)}}
}
```

After replacing all the for loops by sapply function we observed improvement over the basic cross validation but a performance reduction compared to the provious one, from 33450ms without optmisation we did the cross validation in only 31240ms for the same task and the same 1000 observations with vectorization. (the corresponding profiling file is full_opimized_cv.Rproviz).

The revelent profiling scripts can be find in the zip file.

6 Parallel computing

In order to parallelise our code know we can use the R package parallel and use the paralelised version of the sapply function. But before we have to set the cores settings :

```
library(parallel)

# Calculate the number of cores
no_cores <- detectCores() - 1

# Initiate cluster
cl <- makeCluster(no_cores)
```

Know we can change our sapply with parSapply functions. Exemple with the gradient function :

```
gradient_new <-  function(x,y,beta) {
  if(length(beta) != ncol(x)){return("The number of values in beta should
    be equal to the number of variable of X")} else{
      if(length(beta)==1){
        u = exp(x * beta) / (1 + exp(x * beta))
        for (i in 1:length(test)) {g = g + test[i]*(u[i]-y[i])}
      }else{
        g= rep(0,ncol(x))
        u = exp(crossprod(t(x),beta)) / (1 + exp(crossprod(t(x),beta)))
        vec = parSapply(cl, 1:nrow(x),function(i){x[i,]*(u[i]-y[i])})
      }
      return(as.matrix(rowSums(vec))) }
}

hessian_new <-  function(x,y,beta) {
  if(length(beta) != ncol(x)){return("The number of values in beta should
    be equal to the number of variable of X")} else{
    if(length(beta)==1){
      u = exp(crossprod(x,beta))  / (1 + exp(crossprod(x,beta)) )
      h = u[i]*(1-u[i]) * crossprod(x, x)}
    else{
      u =exp(crossprod(t(x),beta))  / (1 + exp(crossprod(t(x),beta)) )
      vec = parSapply(cl, 1:nrow(x),function(i){(u[i]*(1-u[i]))*crossprod(x, x)})
      vec =matrix(rowSums(vec), ncol=ncol(x), nrow = ncol(x))
      return(vec)}}
}
```

After replacing all the sapply functions with parSapply functions we observed nice improvement over the vectorized cross validation, from 31240ms without paralelisation we did the cross validation in only 27410ms for the same task and the same 1000 observations with paralelisation. (the corresponding profiling file is parallel_cv.Rproviz).

7 Reproducible experiment

This model is by definition made with some randomness because of the fact that we build our Y according to some probabilities.

```
set.seed(1)
beta <- matrix(c(0.6, -0.01, -2.5, 12, 0.05), 1, 5)
data <- rlogit(100,beta)
Y <- data[,1]
X <- data[,2:dim(data)[2]]
x <- as.matrix(X)
y <- as.vector(Y)
beta <- t(beta)
basic.modelcomparison(x, y, models, beta0)$CV_model
```

```
#  $CV_best_model
#  [1] 5.33
#
#  $best_model
#  [1] 0 1 0 1 0
```

```
basic.modelcomparison(x, y, models, beta0, selection = "cv")
```

```
#  $CV_best_model
#  [1] 5.15
#
#  $best_model
#  [1] 1 1 1 0 0
```

```
basic.modelcomparison(x, y, models, beta0, selection = "cv")
```

```
#  $CV_best_model
#  [1] 5.42
#
#  $best_model
```

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

Here as you can see we can't recover the exact same result by using the cross validation error criteria.

However if we want to be able to recover the same result we can use the AIC criteria, which moreover is way faster to compute:

```
basic.modelcomparison(x, y, models, beta0, selection = "AIC")
```

```
#  $AIC_best_model
#  [1] -204.364
#
#  $best_model
#  [1] 1 1 1 0 0
```

```
basic.modelcomparison(x, y, models, beta0, selection = "AIC")
```

```
#  $AIC_best_model
#  [1] -204.364
#
#  $best_model
#  [1] 1 1 1 0 0
```

Lets show what is possible to do with my functions :

```
loglikl(x, y, beta)
```

```
#  [1] 6.574147
```

```
gradient(x, y, beta)
```

```
#           X1          X2          X3          X4          X5
#   1.03668773 -0.63860661  1.62997612  0.03289274 -0.62969332
```

```r
hessian(x, y, beta)
```

```
#             X1         X2         X3         X4         X5
#  X1 231.216140  -1.415948   5.744220 -11.084371  42.411541
#  X2  -1.415948 259.523864 -14.179244 -16.228840  43.195152
#  X3   5.744220 -14.179244 302.341705  32.524387   9.814404
#  X4 -11.084371 -16.228840  32.524387 278.482595   7.249212
#  X5  42.411541  43.195152   9.814404   7.249212 386.765444
```

```r
beta0 = c(1,0.5,-2,11.5,0.5)
rhapson_newton(x, y, beta0,0.1)
```

```
#           [,1]
#  X1  0.9932735
#  X2  0.4968233
#  X3 -2.0073226
#  X4 11.5005971
#  X5  0.4981069
```

```r
basiv.cv(x,y, beta0)
```

```
#  [1] 5.46
```

```r
basic.modelcomparison(x, y, models, beta0, selection = "AIC")
```

```
#  $AIC_best_model
#  [1] -204.364
#
#  $best_model
#  [1] 1 1 1 0 0
```

```r
basic.modelcomparison(x, y, models, beta0, selection = "Rsqared")
```

```
#  $R_squared_best_model
#  [1] -0.01121795
```

```
#
#  $best_model
#  [1] 0 1 0 1 0
```

```
basic.modelcomparison(x, y, models, beta0, selection = "cv")
```

```
#  $CV_best_model
#  [1] 5.29
#
#  $best_model
#  [1] 1 1 1 0 0
```

```
basic.modelselection(x, y, "backward", beta0)$CV_model
```

```
#  $CV_best_model
#  [1] 5.47
#
#  $best_model
#  [1] 1 1 1 1 1
```

```
basic.modelselection(x, y, "baCKward", beta0)$CV_model
```

```
#  $CV_best_model
#  [1] 5.4
#
#  $best_model
#  [1] 1 1 1 1 1
```