MASTER FOR SMART DATA SCIENCE

Parallel Computing with R

Project report presented by:

Adrien Chaillout-Morlot

Alessio Crisafulli Carpani

Professor:

Matthieu Marbac-Lourdelle

Année scolaire 2022/2023

Contents

1 Introduction

The purpose of this project is to implement in **R** a procedure for selecting the variables in the logit model (without using the **glm** function). The procedure for selecting the variables is a **stepwise search** which optimizes the prediction error estimated by *cross-validation*.

The project is divided in two parts. In the first part we define the basic functions needed to perform correctly the model. In order to check the consistency of our results we will check the results with the *base R* **glm** function. In the second part instead we will try to optimize our functions, where possible using code profiling and parallel computing.

2 Logistic Regression

Let $\left(X_1^\top, Y_1\right), \ldots, \left(X_n^\top, Y_n\right)$ be observed independent copies of the random vector

$$\left(X^\top, Y\right) \text{ with } \mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \ldots & x_{1,p} \\ 1 & x_{2,1} & x_{2,2} & \ldots & x_{2,p} \\ \ldots & & & & \\ 1 & x_{n,1} & x_{n,2} & \ldots & x_{n,p} \end{bmatrix} \in \mathbb{R}^{n \times (p+1)} \text{ and } Y \in \{0, 1\}.$$

The distribution of Y given $X = x$ is assumed to be a logit model, such that

$$\mathbb{P}(Y = 1 \mid X = x) = \frac{e^{x^\top \beta}}{1 + e^{x^\top \beta}} \text{ and } \mathbb{P}(Y = 0 \mid X = x) = 1 - \mathbb{P}(Y = 1 \mid X = x),$$

where $\beta \in \mathbb{R}^p$ is the vector of the model parameters.

2.1 Generate observations from a logit model

Within the script *"GenerateData.R"*, we define a function rlogit which generate observations from a logit model and from random coefficients, the results are then stored in a list. However, for the reproducibility of the example and to control the expected behaviour of our functions in order to check the consistency of our model selection function, we have defined our parameters calling another function rlogit _with_param.

```
miceadds::source.all(path = "scripts", print.source = FALSE)
set.seed(123)
sample <- rlogit_with_param(1000, params = c(-1, 0, 1.5, -0.85, 0, 2.3))
```

```
R>  <list>
R>  +-X<dbl [6,000]>: 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
R>  +-Y<dbl [1,000]>: 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, ...
R>  \-params<dbl [6]>: -1, 0, 1.5, -0.85, 0, 2.3
```

2.2 Maximum Likelihood Estimation

In this section we will implement the function basic.mle which takes as input the covariates and the dependent variable. This function returns the maximum likelihood estimator (**MLE**) $\hat{\beta}$ defined by

$$\hat{\beta} = \arg\max_{\beta} \sum_{i=1}^{n} \ell\left(y \mid X; \beta\right) \text{ with } \ell\left(y \mid X; \beta\right) = y_i \left(\beta X^T\right) - \ln\left(1 + e^{\beta X^T}\right).$$

The problem of maximization does not admit a closed form solution. However, the MLE can be estimated by the **Newton-Raphson** algorithm. Starting from the initial point $\beta^{[0]}$, the algorithm iterates until the converge is reached. Its iteration $[i]$ is defined by:

$$\beta^{(i+1)} = \beta^{(i)} - H^{-1}\left(\beta^{(i)}\right) \nabla f\left(\beta^{(i)}\right)$$

where ∇f is the *gradient* of the log-likelihood, the vector of its partial derivatives and H is the *Hessian* of f, its matrix of second partial derivatives:

$$\begin{aligned} \nabla f(\beta) =& \frac{\partial l(\beta)}{\partial \beta} = \sum_{i=1}^{n} x_i \left(y_i - p\left(x_i; \beta\right)\right) \\ H(\beta) =& \frac{\partial^2 l(\beta)}{\partial \beta \partial \beta^\top} = -\sum_{i=1}^{n} x_i^\top x_i p\left(x_i; \beta\right)\left(1 - p\left(x_i; \beta\right)\right) \end{aligned}$$

Algorithm 1: Newton-Raphson algorithm for MLE

Result: MLE $\hat{\beta}$
Input : Matrix with covariates X, vector of the variable to predict y,
Output: Vector of parameters $\hat{\beta}$

1 Inititalise a vector as guess for the minimum
2 Define threshold
3 **while** *The relative difference between* $f(\beta), f(\beta)^i < threshold$ **do**
4 | Update the iteration by computing gradient and the inverse of Hessian
5 **end**

2.3 Consistency of the N-R algorithm for MLE

```
source("scripts/MLE.R")
rbind(
  "basic.mle" = basic.mle(sample$X, sample$Y),
  "glm" = glm(sample$Y ~ sample$X + 0, family = "binomial")$coeff
) -> table1
```

	sample$X1	sample$X2	sample$X3	sample$X4	sample$X5	sample$X6
basic.mle	-1.02	0.115	1.22	-0.779	0.11	2.07
glm	-1.02	0.115	1.22	-0.779	0.11	2.07

3 Model Selection

In this part, we will implement some functions in order to get the best model (i.e., the subset of the relevant variables of our data) and its estimator of the prediction error, obtained by **cross-validation** for any subset of covariates.

In order to implement this in R, we will need to define three function:

Table 1: Description of the functions implemented in this section

Variables	Input	Output	Description
`basic.cv`	#Folds, Sample	Error	This function returns the estimator of the error of prediction obtained by cross-validation for any subset of covariates by using the MLE of the model
`basic.modelcomparison`	Sample, Set of Models	Best model, Error	This function returns the best model (i.e., the subset of the relevant variables) and its estimator of the prediction error
`basic.modelselection`	Sample	Best model	returns the best model (i.e., the subset of the relevant variables) and its estimator of the prediction error, using a forward approach

3.1 Cross Validation

In order to estimate how accurately our predictive model will perform in practice, we have computed a function which performs **cross-validation** for any different subset of covariates. Cross-validation is a resampling method that uses different portions of the data to test and train a model on different iterations, in order to get an estimation of the prediction error.

Algorithm 2: Cross Validation Pseudocode

Input : List of models with different covariates
Output: Set of best covariates, perfomance of the models

```
Inititalise the list of possible different models
Define initial threshold for error
for every model in the list do
    Subset the data accordingly
    Compute the MLE with basic.mle
    Get the prediction
    Get the estimate of the error
end
return List with model and performance
```

Figure 1: Graphic Representation of CV

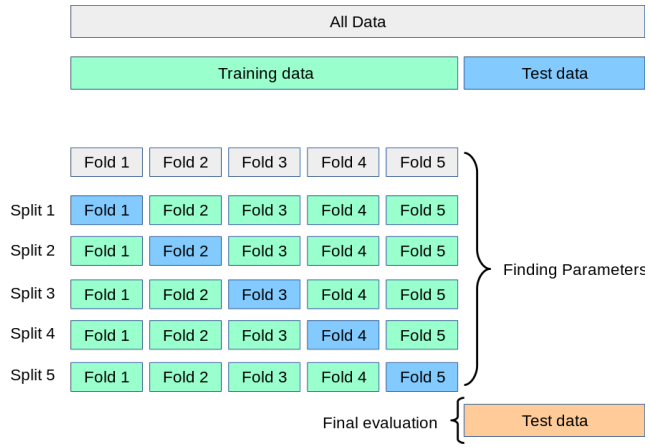


Figure 2: Pseudocode for the cross validation

Below we reported a simple implementation of our `basic.cv` function. In this example we have specified a 5-Fold implementation of the algorithm on our sample. As an estimate of the error of our logistic classifier, we have computed the **accuracy**:

$$Accuracy = \frac{True\ Positive + True\ Negative}{Total\ Population}$$

The results is stored in a list which contains the best model according the accuracy, the list with the parameters in each fold, their respective accuracy and the averaged accuracy.

```
source("scripts/CV.R")
set.seed(4564)
basic.cv(5, sample$X, sample$Y) %>% lobstr::tree()
```

```
R>  <list>
R>  +–best_mle<dbl [6]>: –1.05, 0.14, 1.19, –0.77, 0.16, 2.04
R>  +–params_cv: <list>
R>  | +–<dbl [6]>–1.03, 0.15, 1.21, –0.73, 0.18, 2.07
R>  | +–<dbl [6]>–1.05, 0.14, 1.19, –0.77, 0.16, 2.04
R>  | +–<dbl [6]>–1.09, 0.07, 1.29, –0.76, 0.05, 2.11
R>  | +–<dbl [6]>–1, 0.13, 1.18, –0.88, 0.07, 2.12
R>  | \–<dbl [6]>–0.92, 0.08, 1.22, –0.77, 0.07, 2.03
R>  +–perf_cv<dbl [5]>: 0.82, 0.84, 0.82, 0.81, 0.84
R>  \–accuracy: 0.83
```

3.2 Stepwise Regression: Forward Selection

Forward stepwise selection:

- First, we approximate the response variable y with a constant (i.e., an intercept-only model).
- Then we gradually add one more variable at a time.
- Every time we always choose from the rest of the variables the one that yields the best accuracy in prediction using cross validation, when added to the pool of already selected variables.

```
# Model comparison procedure
set.seed(464)
basic.modelcomparison(sample$X, sample$Y, list(c(1, 2, 3), c(1, 3, 4), c(1, 2, 6), c
    (1, 3, 6), c(1, 3, 4, 5), c(1, 4, 6)))
```

```
R>  $best_model
R>  [1] 1 3 6
R>
R>  $perf_model
R>  [1] 0.808
```

```
# Consistency of the model selection procedure
set.seed(53)
basic.modelselection(sample$X, sample$Y)
```

```
R>  $best_model
R>  [1] 1 3 4 6
R>
R>  $perf_model
R>  [1] 0.834
```

As expected from the data generation, our function did not select the second and fith variables which were generated from parameters set to 0 (also the 4 was close to 0).

4 Optimization

4.1 Code profiling

We started by profiling our functions, understaning which tasks required most computational time. We used the profvis package. The output shows in the top pane the source code, overlaid with bar graphs for memory and execution time for each line of code, whereas the bottom pane displays a **flame graph** showing the full call stack, i.e. the sequence of calls leading to each function.

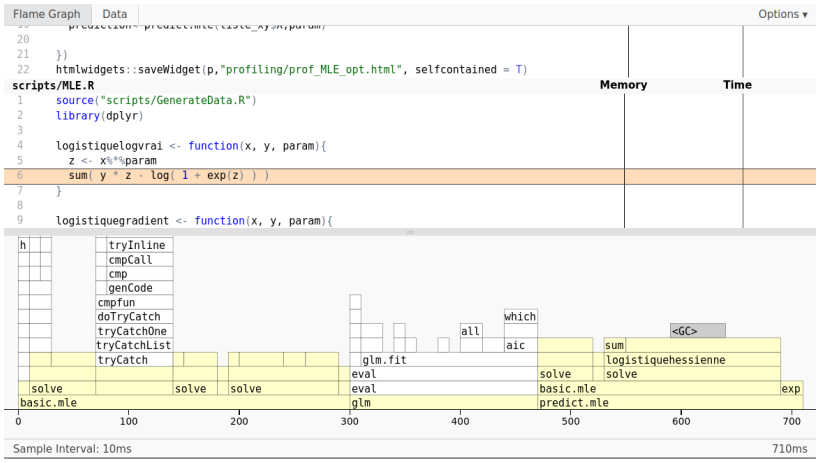


Figure 3: Code profiling of MLE.R

After having understood which were the parts of our codes slowing the functions, we rewrote new functions (in scripts folder, with base name and *_opt*') which optimize the starting ones without the need of parallel computing. Some remarks:

- MLE: we tried using nested sapply instead of nested for loops, however calling it into other functions does not improve the overall. We also tried using matrix operators but did not improve the function. Examples of computing H:

```
z ← x %*% param
z ← exp(z) / ((1 + exp(z))**2)
# 1 option
Hmatrix < sapply(1:ncol(x), function(i) sapply(1:ncol(x), function(j) sum(−z * x
    [, i] * x[, j])))
# 2 option
Hmatrix ← −(t(x) %*% diag(c(z)) %*% x)
# 3 option
Hmatrix ← matrix(NA, ncol(x), ncol(x))
for (i in 1:ncol(x)) {
  for (j in 1:ncol(x)) {
    Hmatrix[i, j] ← sum(−z * x[, i] * x[, j])
  }
}
```

- CV: we used base r subsetting operators instead of functions
- In the model selection functions we call CV_opt instead of CV

4.2 Parallel Computing

When the optimization methods have not apported better result, we implemented **parallel computing** techniques to improve the code. We have used the function mclapply from the package { parallel }.

This approach was used in our basic.cv function, and then it was implemented also within basic.ModelSelection**_par** and we have obtained significant reduction in computational times, see the figure below as an example:

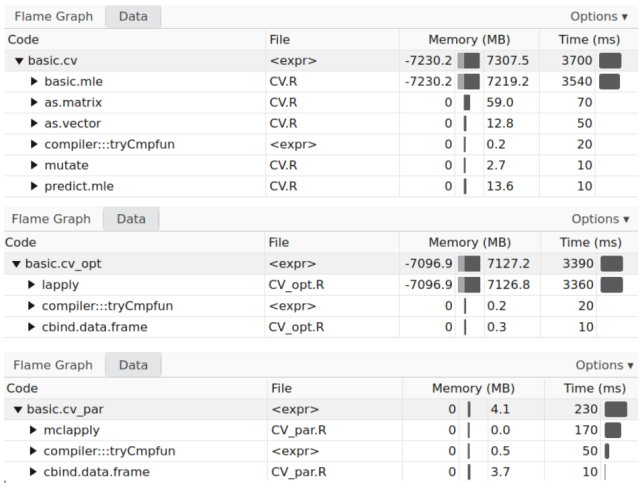


Figure 4: Graphic Representation of CV

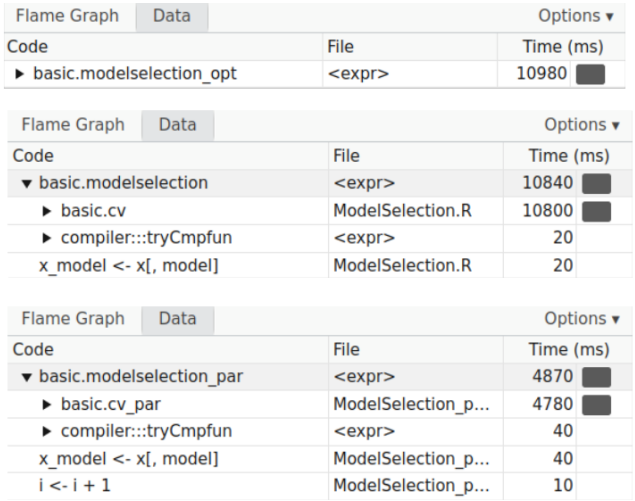


Figure 5: Pseudocode for the cross validation

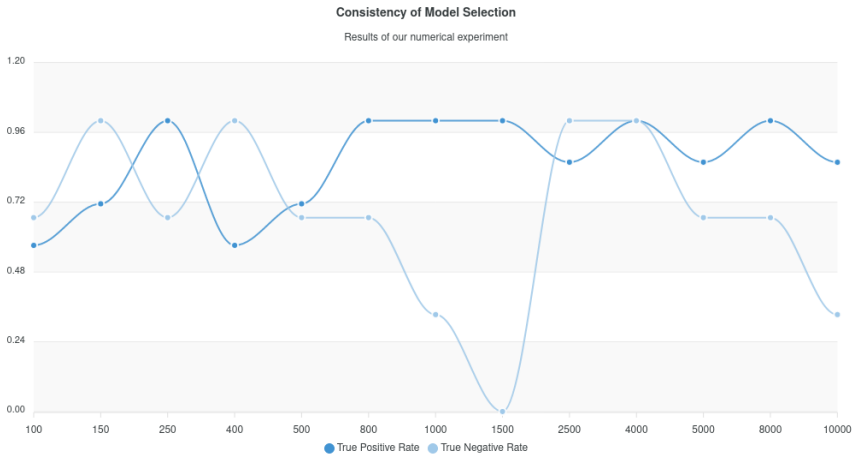
Here is the implementation of our function using mcapply:

```r
# This is the part of our code implementing mclapply
list_cv ← mclapply(1:Kfold, function(i) {
  Xtrain ← as.matrix(data[data$fold != i, -c(1, ncol(data))])
  Xval ← as.matrix(data[data$fold == i, -c(1, ncol(data))])
  Ytrain ← as.vector(data[data$fold != i, 1])
  Yval ← as.vector(data[data$fold == i, 1])

  params ← basic.mle(Xtrain, Ytrain)
  predict_vector ← predict.mle(Xval, params)

  perf ← sum(((predict_vector == Yval) * 1)) / length(Yval)

  param_vec ← params
  perf_vec ← perf
  return(list(param_vec = param_vec, perf_vec = perf_vec))
}, mc.cores = 10)
```

5 Consistency of the model

Illustrate the consistency of the procedure of model selection by a reproducible numerical experiment Besides illustrating the correctness of our modelselection function, as we have done in page 4, we reproduced a numerical experiment to plot the consistency of our implementation. To do this, we wrote a function in *Consistency.R*

```
library(tictoc)
tic()
consistency(vec_obs = c(100, 150, 250, 400, 500, 800, 1000, 1500, 2500, 4000, 5000,
    8000, 10000))
toc() # 162.29 sec elapsed
```



5.1 Numerical Experiment over different CPUs

As the numerical experiment from the step before involved many seconds (162.29 sec elapsed) to provide the result, we thought of improving it using parallel computing with *consistency_par*. The results is shown beow:

Flame Graph | Data | Options ▾

Code	File	Memory (MB)		Time (ms)
▼ consistency	<expr>	-75.5	96.1	51950
▶ basic.modelselection_par	Consistency.R	-75.5	94.9	51890
▶ rlogit_with_param	Consistency.R	0	1.0	30
▶ compiler:::tryCmpfun	<expr>	0	0.1	20
True_Negative	Consistency.R	0	0.0	10

Flame Graph | Data | Options ▾

Code	File	Memory (MB)		Time (ms)
▼ consistency_par	<expr>	0	3.9	310
▶ rlogit_with_param_parr	Consistency_par.R	0	0.9	210
▶ t	Consistency_par.R	0	3.0	70
▶ compiler:::tryCmpfun	<expr>	0	0.1	20
▶ True_Positive	Consistency_par.R	0	0.0	10

Appendix

Project Structure

Below we are reporting the content of our zipped submission file. It contains an R project file (*.Rproj*), the Rmarkdown file used to knit the report and the converted *.pdf* report, along with some folders:

```
.
├── Consistency.R
├── ParallelComputingR.Rproj
├── Parallel_Logistic.Rmd
├── Parallel_Logistic.pdf
├── appunti
├── profiling
├── scripts
├── test
└── utils
```

- The folder *scripts* cotains the scripts with the different functions defined for the different tasks.
- The folder *profiling* contains the results of code profiling of each function
- The folder *test* to show the functions in action
- The folder *utils* it was needed just to compile the report, not useful for the purpose of the project.