Stat243 Final Project: Genetic Algorithm R Package

Côme de Lassus Saint Geniès, Arman Jabbari,
Qingan Zhao, & Mia Zhong

December 13, 2017

Github user name: QinganZhao

1 Introduction

Genetic algorithms mimic the Darwinian natural selection process to solve optimization problems[1]. Every candidate solution corresponds to an individual of a population and is represented by its genetic code, referred to as chromosomes in the following discussion. Each individual's genetic code embodies one candidate solutions to the optimization problem. By breeding among individuals who have genetic code with better fitness in terms of the optimization problem, genetic algorithms allow the population to evolve and become increasingly fit.

This current project aims at implementing a genetic algorithm to select variables in regression problems. The primary function in the package GA, *select*, takes datasets X and Y that look for improvement in regression modeling. It outputs the selected predictors among the candidates, the improved fitness value, and regression coefficients of the selected predictors. While this GA package could be applied to linear regression and GLM problems with detailed default settings on fitness function, population generation and the breeding process, it also guarantees good flexibility for users to input ideal settings for specific problems.

This document is organized as follows. Section 2 for the function structure introduces how the code was modularized and how we approached function programming. In Section 3, we illustrates the results of auxiliary function tests and the application of *select* function with simulated data and real world survey data. Section 4 concludes the contributions of our

[1]Givens, G.H., & Hoeting, J.A. (2012), Computationa statistics (Vol.710). John Wiley & Sons

team members.

2 Code structure

The development of our GA package went through two stages: 1) developing the *select* function and 2) building up the package around the *select* function. To build the *select* function, the algorithm was modularized with functions implementing different tasks, as shown in Figure 1. Next, we test the *select* function with both simulated datasets and data from Japan Social Survey 2010. As the *select* function works well in both scenarios, we conclude our project by completing the package construction around the *select* function.

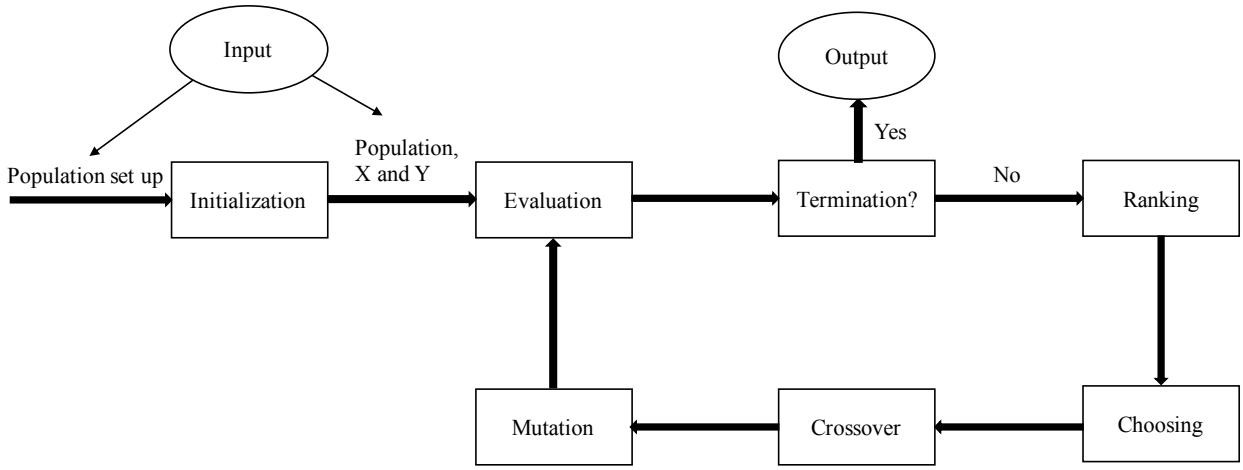

Fig. 1: Module of the *select* function

Input and population set-up The *popInitial* function initializes the first generation population for the genetic algorithm, with each row representing the chromosome of an individual in the population. This function allows users to set up the population size and the rate of zeros in the matrix of chromosomes.

Both the *popInitial* package and the *select* package check and build up the inputs of the genetic algorithm. The population size and gene length should be positive integers. X (matrix) and Y (vector) are the datasets provided by users that look for a better regression model. X and Y should match with each other in dimensions, and the first generation chromosomes is generated based on the number of columns of X.

Evaluation The *evaluation* function takes X, Y and the current generation of chromosomes for fitness evaluation. The regression type is set default as *lm()* and users can alternatively choose the *glm()* function. Meanwhile, the fitness function is set default as AIC, also allowing flexibility if the users want to use other fitness functions. This *evaluation* function outputs a dataframe with two columns, the first containing strings of chromosomes and the second containing the correspondence AIC fitness value of each functions with predictors selected by the chromosomes. The output format of chromosomes are designed in string format for the convenience of *rankSelection* and *choosing* functions. As explained in the section for *breed* function, *breed* will transform the chromosomes in string characters back to vectors for conducting *evaluation* again.

The *evaluation* function is designed to return warnings when it detects NAs in data or the correlations between X and Y are too strong. The potential problems with the input X and Y are the existence of NAs and the already strong linear correlations. On the one hand, as the *lm()* and *glm()* functions defaultly conduct *na.omit* for NAs in the datasets, the evaluation result would warn the users that the observations with NAs are omitted in the function. On the other hand, if the variables in X already significantly correlates with Y, the AIC function would return -Inf as fitness level, indicating an overfitting scenario for the selection.

The *superEvaluation* function employs the *foreach* function to allow users parallel the *evaluation* over multiple cores when the number of candidate predictors are large or the criterion function is computationally expensive. The *superEvaluation* function is turned off at default setting and users could utilize it when calling the *select* function.

Termination criterion As the literature suggest[2], the stopping criteria for genetic algorithms are usually categorized into three types:

1. Stop when the upper limit on the number of generations are reached;
2. Stop when the upper limit on the number of evaluations of the fitness function is reached;
3. The chance of achieving significant changes in the new generation is excessively low.

In this project, we adopt the third type of stopping criteria which is adaptive to the consequent results of iterative selection process as it doesn't require prior knowledge about the

[2]Safe, M., Carballido, J., Ponzoni, I., & Brignole, N.(2004). On stopping criteria for genetic algorithms. Advances in Artificial Intelligence-SBIA 2004, 405-413.

specific problems provided by the users. After the iteration time exceeds the minimum level (min_iter), the *select* function compares the best fitness value of the new generation with the mean of best fitness values of the last min_iter generations. The loop of genetic breeding stops when the two values are approximately equal to each other or the iteration time exceeds the maximum level (max_iter). Both (min_iter) and (max_iter) are arguments of the *select* function that are open to user settings.

Rank selection and choosing parents The *rankSelection* function takes an input data frame with one column consisting of string chromosomes and the other column consisting of AIC fitness values. The function provides two approaches to prepare rank genes for selection: users can either assign probabilities to chromosomes based on the ranking of their fitness level (set as default) or assign selection probabilities to chromosomes proportionate to the fitness values. Chromosomes with lower AIC fitness values correspond to better fit linear regression models. The default setting of *rankSelection* function uses the formula suggested by Givens:

$$probability = \frac{2(popNum + 1 - i)}{popNum * (popNum + 1)}$$

where i is the ranking of the corresponding chromosome.

In order to pair parents from the ranked dataset, the *choosing* function restricts the size of generated population (popNum) to be even numbers. This function takes the input dataframe with one column of string chromosomes and the other of ranked probabilities for parent choosing. Two options are available for choosing parents from the population: user can either choose both parents based on the results after ranking (rankBased = TRUE as default), or choose one parent based on ranking and the other uniformly. The output of the *choosing* function is a dataframe with two columns, each row representing parent chromosomes in the format of string characters.

Breed (crossover and mutation) The *breed* function combines the *crossover* function and the *mutation* function ,with the additional functionality of transforming data types. The *select* function calls the *breed* function directly in the while loop.

- Transformation between strings and vectors
  We choose to build the data type transformation functionality in to the *breed* function because we want to keep the convenience and intuitiveness for previous *rankSelection* and *choosing* functions as well as for the *crossover* and *mutation* functions. The *breed*

function takes the chosen parent dataframe, which contains two columns of string chromosomes, and transform the parent chromosomes into vectors. Then, *breed* applies *crossover* and *mutation* upon the vectorized chromosomes to give births to the new generation of genes. The vectorized chromosomes makes it easier for random cuts and mutations, keeping the data format consistent for a new round of evaluation.

- Crossover randomness
  The *crossover* function takes the two parent chromosomes in vectors and create two new genes from them by cutting at random points and crossover. This function allows users to set numbers as cut points (set to 1 as default). For more crossing possibilities, the *crossover* function will only keep unique indexes from the generated nCuts indexes. For example, when nCuts = 3, it's possible that 2 of the 3 generated indexes are duplicated. In this case, the *crossover* function would only make 2 cuts for crossover.

- Enforcement and flexibility in mutation
  The *mutation* function conducts change in chromosomes based on the randomness probability provided by users (set as 0.01 in default). On top of this basic functionality, if the genes of any chromosomes are identically 0, we force a mutation by changing one random gene to 1 before evaluation in the next round. Otherwise, the bred generation would not be able to pass the evaluation. We further advance the flexibility of the mutation process in the *select* function. While users can set the probability for mutation, they are also able to curve the mutation probability as the iterations proceed by turning on "mut_pCurve". In this curving case, the *select* function will start the iterations with a higher mutation probabilities, moving toward the lower bound, the set mutation probability. All these features would facilitate the mutation functionality to ensure that the generated population structure won't break the iterations.

Use the package Users should be able to import the GA package with the following code (note that dplyr and devtools packages should be installed first):

```
devtools::install_github('QinganZhao/GA')
library(GA)
```

To utilize the parallelization functionality, users should install packages parallel, doParallel and foreach to implement parallelized evaluation.

3 Tests with datesets

Auxiliary functions

The *test_secondary.R* file contains code for testing auxiliary functions with simulated data. We test whether the *evaluation* function and the *superEvaluation* function return correct results with both simple data and complex data. We also make sure that the *crossover* function would return correct pairs of chromosomes regardless of randomness and the length of chromosomes. With the *breed* function and the *popInitial* function, we test whether two identical generations would be produced with no mutation and whether the number of 1-alleles remains constant with no mutation. Lastly, we check if the *rankSelection* function assigns correct probabilities to ranked chromosomes.

Primary function on simulated data

We also conduct rigorous tests with the primary *select* function. First, we test if *select* is able to return errors with user-provided data. It should be able to tell if the dimensions don't match, datasets contains numerical elements or the number of iterations are not valid. The *select* function should show warnings if some fitness values goes to negative infinitive during the iterations, indicating an overfitting scenario. Also, we test if the function can find best predictors with NAs in input datasets.

Second, we test if the *select* function returns best predictors with simulated datasets under four conditions that captures the default setting and its alternatives based on users' choices. The four conditions default settings, probabilities assignment based on AIC fitness values instead of ranking, parallelized evaluation and *glm()* regression type. For the default setting, we generate a dataframe with obvious correlations between variables and the other dataframe that yields a less fit regression model to reinforce the execution results of the *select* function.

Here are the 4 tests for selecting predictors using the primary function based on simulated data:

```
test_that('finds best predictors with default', {
  #obvious dataframe
  X <- data.frame(matrix(ncol = 20, nrow = 20))
  X[,1] <- 1:20
```

```
  for (i in 2:20){X[,i]<- rep(i,20)+runif(20,0,1)}
  S <- select(X,1:20,usingRank = FALSE)
  names(S$coefficients) <- NULL
  correct <- rep(0,length(S$coefficients))
  correct[2] <- 1
  expect_true(all.equal(S$coefficients,correct,tolerance = 10^-12))
  #less obvious regression
  X <- data.frame(matrix(ncol = 20, nrow = 20))
  for (i in 1:20){X[,i]<- rep(i,20)+runif(20,0,1)}
  S <- select(X,1+2*X[,1]-3*X[,2]+0.0001*X[,3])
  names(S$coefficients) <- NULL
  correct <- rep(0,length(S$coefficients))
  correct[1:4] <- c(1,2,-3,0.0001)
  expect_true(all.equal(S$coefficients,correct,tolerance = 10^-12))
})

##  Selected predictors: X1 X2 X3 X4 X5 X8 X9 X12 X15 X20
##  Fitness value: -1316.267
##  Selected predictors: X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X12 X13 X14 X15 X16 X17 X18 X19
##  Fitness value: -Inf

test_that('finds best predictors without using rank', {
  X <- data.frame(matrix(ncol = 20, nrow = 20))
  for (i in 1:20){X[,i]<- rep(i,20)+runif(20,0,1)}
  S <- select(X,1+2*X[,1]-3*X[,2]+0.0001*X[,3],usingRank = FALSE)
  names(S$coefficients) <- NULL
  correct <- rep(0,length(S$coefficients))
  correct[1:4] <- c(1,2,-3,0.0001)
  expect_true(all.equal(S$coefficients,correct,tolerance = 10^-12))
})

##  Selected predictors: X1 X2 X3 X4 X5 X7 X9 X11 X14 X19
##  Fitness value: -1306.102

test_that('finds best predictors with parallelization', {
  X <- data.frame(matrix(ncol = 20, nrow = 20))
  for (i in 1:20){X[,i]<- rep(i,20)+runif(20,0,1)}
```

```
  S <- select(X,1+2*X[,1]-3*X[,2]+0.0001*X[,3],useParallel = TRUE)
  names(S$coefficients) <- NULL
  correct <- rep(0,length(S$coefficients))
  correct[1:4] <- c(1,2,-3,0.0001)
  expect_true(all.equal(S$coefficients,correct,tolerance = 10^-12))
})

##  Selected predictors: X1 X2 X3 X4 X5 X7 X8 X9 X10 X14 X15 X16 X17 X19
##  Fitness value: -Inf

test_that('finds best predictors with glm', {
  X <- data.frame(matrix(ncol = 20, nrow = 20))
  for (i in 1:20){X[,i]<- rep(i,20)+runif(20,0,1)}
  S <- select(X,1+2*X[,1]-3*X[,2]+0.0001*X[,3],reg = 'glm')
  names(S$coefficients) <- NULL
  correct <- rep(0,length(S$coefficients))
  correct[1:4] <- c(1,2,-3,0.0001)
  expect_true(all.equal(S$coefficients,correct,tolerance = 10^-12))
})

##  Selected predictors: X1 X2 X3 X6 X9 X13 X15 X18 X19
##  Fitness value: -1339.372
```

Primary function on real world data

Besides simulated datasets for auxiliary functions and the *select* function, we also test the *select* function with a real world dataset from Japan Social Survey 2010. We test if the *select* function is able to select the strong predictors from a large pool of predictors for **income** level (taken log) for individual respondents in JGSS 2010. Variables in dataframes X and Y are listed in Table 1.

We choose the 8 strong predictors based on theoretical and empirical studies around determinants and correlates of income levels, especially studies on the Japanese society[3][4][5]. Weak

[3]Tachibanaki, T. (2009). Confronting income inequality in Japan: A comparative analysis of causes, consequences, and reform.MIT Press Books, 1.Chicago

[4]Bauer, J., & Mason, A. (1992). The distribution of income and wealth in Japan. Review of Income and wealth, 38(4), 403428.

[5]Griliches, Z., & Mason, W. M. (1972). Education, income, and ability. Journal of political Economy,

Table 1: Variables in dataframes X and Y

Variable name	Description	Variable name	Description
	Y − observations		
SZINCOMA	Annual income (log)		
	X − predictors		
	Strong predictors		
SEXA	Gender	XJOBIWK	Employment status
AGEB	Age in 2010	XXLSTSCH	Last school attended
MARC	Marital status	JOINUNI	Join labor union
TP5LOC15	Place of Residence at Age 15: Size of Municipality	PPLSTSCH	Last school father attended
	Weak predictors		
PPLVTG	Father still alive	FQSPORT	Frequency of regularly sports
DAY	Date of interview (1-31)	MONTH	Month of interview (1-12)
DAYB	Date of self-admin questionnaire collected	BD3SAFTY	Opinions on Governments Expenditure on National Safety
Q4NOPWR	View on Citizens Influence on Politics	OPCO2EM	View on reduction of CO_2 Emission
Q4NOCCMG	View o Obligation to Have Children	OPTEED	View on the Timing to Start English Education
ST5HLTHY	Satisfaction with Health Condition	ST5LIFEY	Satisfaction with Family Life
DORL	Practice Religion	Q4DIVOK	View on Divorce
OP4NAME	View on Change of Surname at Marriage	WLACCORG	Acceptance of Organ Donation

predictors are less directly related to income level, including 1) opinions and personal views on social issues and 2) personal habitual facts.

The *select* function passes the test if it returns 5 out of 8 strong predictors. The variable pool is quiet large in this example, and the selected strong predictors themselves don't make up a strong linear fit for Y (the r square of the linear model is quiet small). Moreover, adding weak predictors complicates the correlations among these variables and increases noise. Therefore, it's possible that the effects of some strong predictors are blurred by other weak ones.

Test code is shown as follows:

```
test_that('test if most strong predictors in a real-world dataset can be selected', {
  download.file('https://github.com/QinganZhao/Data-Science/blob/master/database/JGSSdat
  load('jp')
  jpY <- jp[,c("SZINCOMA")]
  Yjp <- as.matrix(log(jpY))
  Xjp <- jp[,c("SEXA", "AGEB","MARC", "XJOB1WK", "XXLSTSCH", "JOINUNI",
               "PPLVTG","DAY", "FQSPORT","DOSMOKEX","BD3SAFTY","Q4NOPWR",
               "OPCO2EM","APPCCSXB","Q4NOCCMG","OPTEED", "ST5HLTHY",
               "DAYB", "MONTH", "DORL", "Q4DIVOK", "OP4NAME", "WLACCORG",
               "ST5LIFEY", "TP5LOC15", "PPLSTSCH")]
  Xjp <- as.matrix(Xjp)
  StrongPredictors <- c("predictor.SEXA", "predictor.AGEB", "predictor.MARC",
                        "predictor.XJOBIWK", "predictor.XXLSTSCH",
                        "predictor.JOINUNI","predictor.TP5LOC15",
                        "predictor.PPLSTSCH")
  lmSelect <- select(Xjp, Yjp, reg = 'lm')
  glmSelect <- select(Xjp, Yjp, reg = 'glm')
  lmResult <- names(lmSelect$coefficients)
  glmResult <- names(glmSelect$coefficients)
  #check if at least 5/8 of the known strong predictors have been selected
  checkLm <- sum(StrongPredictors %in% lmResult) >= 5
  checkGlm <- sum(StrongPredictors %in% glmResult) >= 5
  expect_equal(checkLm, TRUE)
```

```
  expect_equal(checkGlm, TRUE)
})

##  Selected predictors: SEXA AGEB MARC XJOB1WK XXLSTSCH JOINUNI PPLVTG DOSMOKEX Q4NOCCM
##  Fitness value: 13575.92
##  Selected predictors: SEXA AGEB MARC XJOB1WK XXLSTSCH JOINUNI ST5HLTHY MONTH DORL WLA
##  Fitness value: 13576.99
```

The test results show that our algorithm was able to select out the majority of strong predictors in the outcome regression model. The fitness level after selection decreased from the first evaluation outcome yet remained very high compared to simulated data. We argue that this high AIC level should come from the fact that it was hardly a good fit for the income variable from the predictors in this dataset X.

The change of the fitness value in the above test can be drawn, as shown in Figure 2.

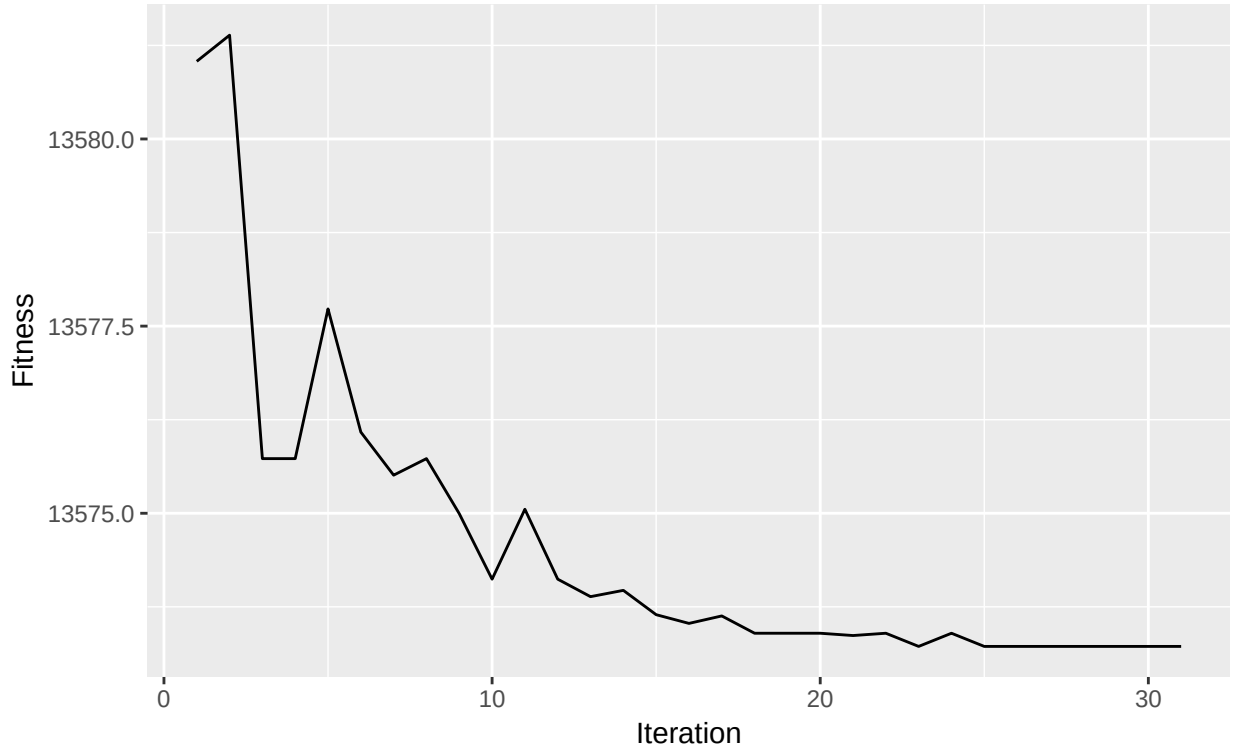

Fig. 2: Fitness value of the test on Japan Social Survey 2010 dataset

We also implemented the algorithm on *mtcars* data provided in Rstudio, and checked the fitness value as shown in Figure 3 to further confirm that the function works.

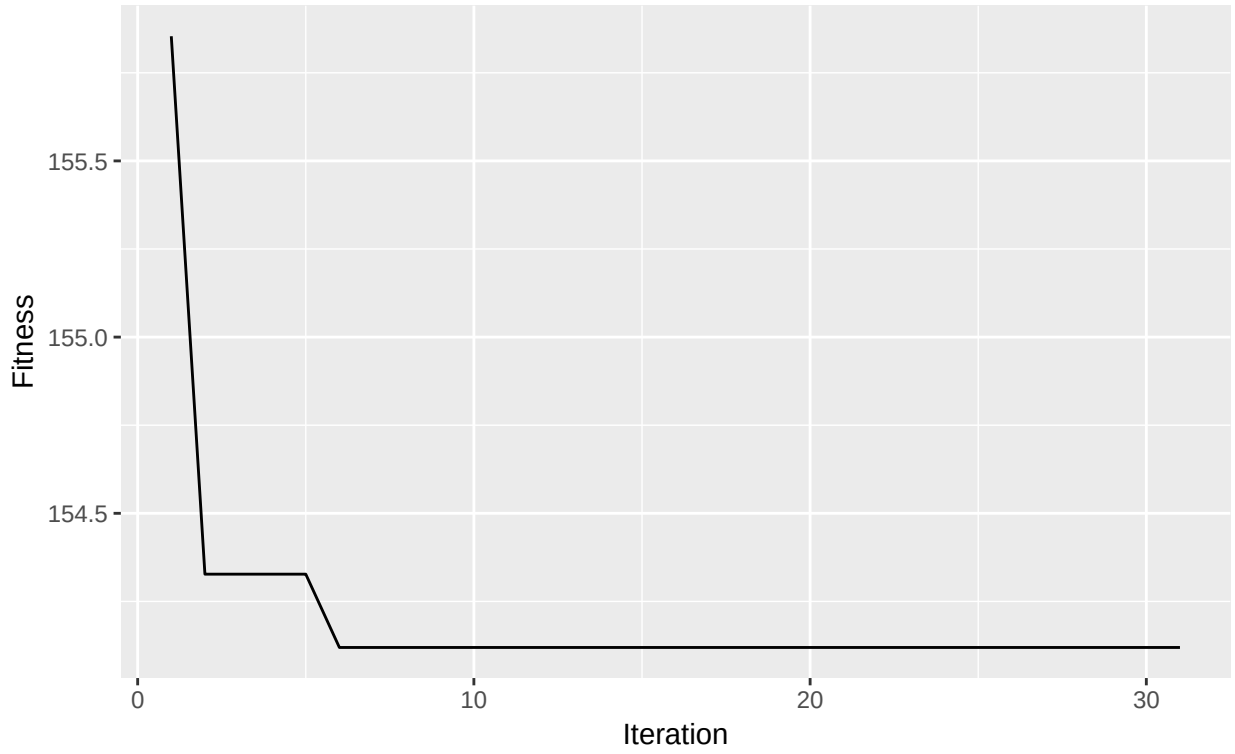

Fig. 3: Fitness value of the test on mtcars

4 Teamwork and contributions

As we divided the project into two phases, we distributed the tasks among members accordingly. In terms of function writing, **Qingan** was in charge of writing *inputs* tests and the *initialization* function for the starting population. **Mia** wrote the function for *fitness* evaluation and build the while loop into the select function for termination (with help from Qingan). **Côme** wrote the functions for *ranking* and *crossover* and **Arman** was in charge of the function for *choosing* parent chromosomes from ranked results and the function for *mutation*. At the end of the function writing phase, **Qingan** combined the *crossover* and *mutation* functions into the *breed* help transform the chromosome data back and forth between string format and vector format. **Qingan** also added the *superEvaluation* for allowing users to conduct evaluation with parallelization.

At the stage of testing and finalizing the package, **Qingan** built up the package, **Arman** wrote the help files, **Côme** and **Mia** conducted the tests for our algorithm and **Mia** and

Qingan wrote the document as the solution to the project. Also, **Qingan** and **Arman** were responsible for Git management during the process. The summarized contributions are shown in Figure 4.

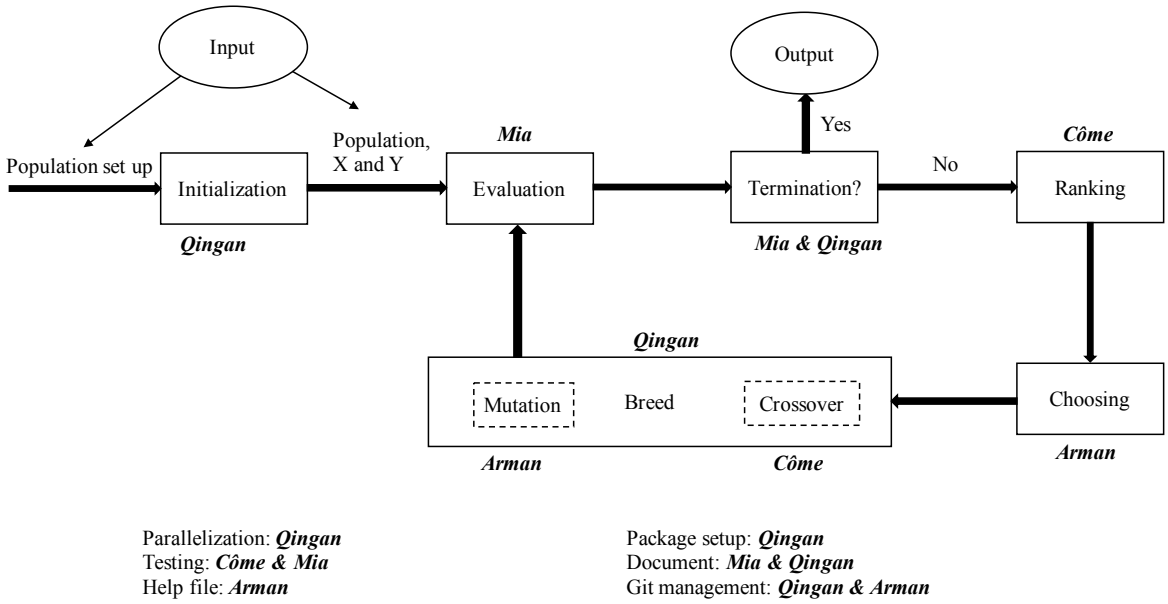

Fig. 4: Contributions of team members