Stat243 Final Group Project

Cameron Adams, Yuwen Chen, Weijie Xu, Yilin Zhou

December 2017

Please note that the Github username that we are using for this project is "yzhou63".

I. Approach Taken

To perform feature selection in regression problems using genetic algorithm, we conduct the following steps sequentially:

1. Initialization: create the first generation – generation_t0 (with function "generate_founders")
2. Evaluatation: assess the fitness of each chromosomes or models, and rank the values of fitness in the population (with function "evaluate_fitness")
3. Selection, crossover, and mutation: select the more fit "parents" and generate a new generation using possible crossover and mutation (with function "create_next_generation")
4. Looping and termination: repeat step 2 and 3 until either the maximum number of iterations is reached or the algorithm converges (with the primary function–"select")

Step 1: Initialization

We randomly create the first generation with the function "generate_founders". As a generation size of P ranging from C and 2C is recommended for binary codings of chromosomes/predictive models (where C is the length of the chromosome or the number of predictors), we calculate P as 2C. Also, since P is less than 200 in most real applications, we set 200 as the upper bounds of P in this case (if P exceeds 200, we would print a warning to the user). We then randomly sample 1.2*C*P 0's and 1's, splitting them up into the P chromosomes/models (note that we sample 1.2*C*P instead of C*P as we want to have enough data so that each chromosomes/model could be unique). We also add two additional samples to our data, where the first sample has the last design variable as 1 and the other design variables as 0, and the second sample has the last design variable as 0 while the others being 1. In this way, we ensure that each gene/predictor appears in at least one chromosome/model, but not all chromosomes/models. After selecting chromosomes with unique genes or models with unique design variables and ensuring that we have at least one 1's in each choromosome or model, we obtain our first generation–generation_t0.

```
###############
# auxiliary function I
###############

# This function will initiative the founding chromosomes
# the output is a randomly created first generation

# the inputs are:
# X is a matrix that contains the predictors in the input dataset
# start_chrom is the user-defined size of a generation


generate_founders <- function(X, start_chrom) {

    # number of predictors ---------------
```

```r
    C <- dim(X)[2]

    # number of founders ----------------
   if (is.null(start_chrom)) {
        P <- 2 * C
        if (P > 200) {          #check for the maximum chrom
            P <- 200
            }
        if (P %% 2 != 0) {      #check for even number of parents
            P <- P - 1
        }
    } else {
      if (start_chrom > 200) cat("Warning: P > 200, algorithm may require lots of running time")
      P <- start_chrom  #start_chrom is the number of chroms defined by the user
      }

    #randomly generate founders ----------------
    geneSample <- sample(c(0, 1),
                         replace = TRUE,
                         size = ceiling(1.2 * C * P))

    #update geneSample to make sure that each gene/variable
    #will exist in at least one chrome, but not all.
    geneSample <- c(rep(0, C - 1), rep(1, C), 0, geneSample)

    #create a first generation
    indices <- seq_along(geneSample)
    first_gen_raw <- split(geneSample, ceiling(indices / C))

    generation_t0 <- matrix(unlist(unique(first_gen_raw)[1:P]),
                       ncol = C, byrow = TRUE)

    generation_t0 <- generation_t0[apply(generation_t0[,], 1,
                                     function(x) !all(x == 0)), ]

    return(generation_t0)
}
```

Step 2: Evaluation

We then evaluate the fitness of each chromosome/model using the function "evaluate_fitness". We first define an auxiliary function named "rank_objective_function", which takes its inputs as the values of the fitness for each model and whether the user requires minimization. It will return each model's fitness and its corresponding rank as the output. Note that if the user requires the fitness function to be those such as AIC or BIC, then the ranking will be based on the negative of these values, and if functions such as log-likehood is the objective criterion, then their positive values will be used.

```r
##############
# auxiliary functions II
##############

# This function will give the rank of the fitness of a model
# based on the targeted objective function and whether a minimization
# or maximization is desired for the objective function

# obj_fun_output is a numeric vector containg the objective function output
# for each chromosome for ranking
# minimize a logical value indicating whether to rank according to
# minimization or maximaziation optimization, default is minimize

rank_objective_function <- function(obj_fun_output, minimize) {

    P <- length(obj_fun_output)

    if (isTRUE(minimize)) {
        r <- base::rank(-obj_fun_output, na.last = TRUE, ties.method = "first")
    } else {
        r <- base::rank(obj_fun_output, na.last = TRUE, ties.method = "first")
    }

    return(cbind(chr = 1:P, parent_rank = r, obj_fun_output))
}
```

We then evaluate the fitness of each model through using the function "evaluate_fitness". We consider several variations depending on the users' inputs: whether our variable selection is based on linear regression or GLM, what we use for our fitness function (AIC, BIC, log-likelihood, a user-defined function, etc.), and whether we would use parallel processing (if we have more than one cores available, then we would use parallel processing). We return the rank and the value of the fitness function as our output for each model.

```r
# This function will evaluate and rank the fitness of each model based on
# targeted objective function
# the default is a linear model assessed with AIC
# the output is the value and rank of the fitness function for each model

# the inputs are:
# generation_t0 is a matrix of parent chromosomes to be evaluated
# the columns correspond to predictors/genes and rows correspond to parents/chromosomes
# Y is a vector of response variable
# family describes the distribution of errors and the link function used
# if family is Gaussian, we will fit the data using lm
# otherwise, we will implement the corresponding glm
# objective_function is the selected criteria to assess the fitness of a model
# nCores is an integer indicating the number of parallel processes
# to run when evaluating fitness
# rank_objective_function a function that ranks parents by their fitness
# as determined by optimize criteria

evaluate_fitness <- function(generation_t0, Y, X,
                             family,
                             nCores, minimize,
                             objective_function,
                             rank_objective_function) {

    #number parent chromosomes
    P <- dim(generation_t0)[1]

    ######
    #evaluate and rank each chromosome with selected objective function
    ######

    # serial ----------------
    if (nCores == 1) {

        # lm ----------------
        if (family == "gaussian") {
            obj_fun_output <- sapply(1:P, function(i) {
                mod <- stats::lm(Y ~ X[, generation_t0[i, ] == 1])
                return(objective_function(mod))
            })
            # glm ----------------
        } else if(family != "gaussian") {
            obj_fun_output <- sapply(1:P, function(i) {
                mod <- stats::glm(Y ~ X[, generation_t0[i, ] == 1], family = family)
                return(objective_function(mod))
            })
        }

        # parallel ----------------
    } else if (nCores > 1) {

        # lm ----------------
        if (family == "gaussian") {
            obj_fun_output <- unlist(parallel::mclapply(1:P, function(i) {
                mod <- stats::lm(Y ~ X[, generation_t0[i, ] == 1])
                return(objective_function(mod))
            }, mc.preschedule = TRUE, mc.cores = nCores))
            # glm ----------------
        } else if(family != "gaussian") {
            obj_fun_output <- unlist(parallel::mclapply(1:P, function(i) {
                mod <- stats::glm(Y ~ X[, generation_t0[i, ] == 1], family = family)
                return(objective_function(mod))
            }, mc.preschedule = TRUE, mc.cores = nCores))
        }
    }

    # rank ----------------
    parent_rank <- rank_objective_function(obj_fun_output, minimize)

    # return rankings ----------------
    return(parent_rank)
}
```

Step 3: Selection, Crossover, and Mutation

In the third step of our algorithm, we create a new generation using selection, crossover and mutation through the corresponding functions and the function "create_next_generation". The population size of a future generation remains the same as the initial generation, so we would still have P individuals or models in the next generation.

We consider several methods for selection, crossover and mutation.

For selection, we select the first parent by the selection probability and the other parent randomly. The probability of selection is calculated as $\phi_i = \frac{2*r_i}{P*(P+1)}$. The algorithm can also implement the user-defined selection method through using match.fun().

A pair of parents has a probability of 0.8 to crossover, and we define three possible methods:

First method: uses three-point crossover, where we choose three non-overlapping points within the length of the chromosome or the number of the design variables in each model. We swap the elements between these points and render two children.

Second method: defines a crossover probability based on the ranks of fitness of each parent, which gives a parent with a higher rank higher probability of passing its genes to the offsprings. Specifically, this crossover probability is a weighted average of the two parents, where the weight for a parent is defined as the proportion of his rank in the summation of the two parents' ranks.

Third method: passes the same variables between parents directly to the offsprings and only considers crossover between the non-concordant variables of parents. Deciding between these non-concordant variables based on a selection probability, which gives a parent with higher rank a higher probability of retaining a variable.

For mutation, we will either use the user-defined mutation probability or the default, which is $\frac{1}{P\sqrt{C}}$.

```r
##############
# auxiliary function III
##############

# This function will select optimal parents
# based on the selection probability

# the input is:
# parent_rank is a vector indicating the rank of parents chromosomes,
# ranking order is inverse: parent chromosomes with lowest fitness rank will
# have rank == 1

select_parents <- function(parent_rank) {

    # get number of chromosomes
    P <- length(parent_rank)

    # probability of selection
    phi <- (2 * parent_rank) / (P * (P + 1))

    # select first parent by parent_rank, second random
    parent1 <- sample(1:P, 1, prob=phi, replace = T)
    parent2 <- sample((1:P)[-parent1], 1, replace = T)

    return(c(parent1, parent2))
}

# This function defines three crossover methods

# the inputs are:
# generation_t0 is a matrix of parent chromosomes to be evaluated
# the columns correspond to predictors/genes and rows correspond to
# parents/chromosomes
# parentInd is a vector containing the rank indexes of two parents selected for crossover
# crossover_method is a character string indicating which method of crossover to be used
# the default is method 1, which is a three-point crossover
# pCrossover is a number between 0 and 1 indicating the probability of
# crossover for each mate pair, the default is 0.8
# parent_rank is an integer vector of fitness ranks for parent chromosomes

crossover_parents <- function(generation_t0, parentInd,
                              crossover_method, pCrossover, parent_rank)  {

    # get parent info
    parent1 <- generation_t0[parentInd[1], ]
    parent2 <- generation_t0[parentInd[2], ]
    C <- length(parent1)
    parent1r <- parent_rank[parentInd[1]]
    parent2r <- parent_rank[parentInd[2]]

    if (stats::rbinom(1, 1, pCrossover) == 1 ) {
        if (crossover_method == "method1") {

            #METHOD 1 ----------------
            #multipoint crossover: three crossover points
            cross <- sort(sample(seq(2,(C - 2), 2), 3, replace = F))

            child1 <- c(parent1[1:cross[1]],
                        parent2[(cross[1] + 1):cross[2]],
                        parent1[(cross[2] + 1):cross[3]],
                        parent2[(cross[3] + 1):C])
            child2 <- c(parent2[1:cross[1]],
                        parent1[(cross[1] + 1):cross[2]],
                        parent2[(cross[2] + 1):cross[3]],
                        parent2[(cross[3] + 1):C])

        } else if (crossover_method == "method2") {

            #METHOD 2 ----------------
            #method upweights parent with higher parent_rank high
            childProb <- parent1 * parent1r[1] /
                (parent1r + parent2r) +
                parent2 * parent2r /
                (parent1r + parent2r)
            child1 <- stats::rbinom(C, 1, prob = childProb)
            child2 <- stats::rbinom(C, 1, prob = childProb)

        } else if (crossover_method == "method3") {

            #METHOD 3 ----------------
            #randomly samples non-concordant variables between parents
            # slightly upweights parent selected by prob. proportional to parent_rank
            child1 <- parent1
            child2 <- parent2
            child1[parent1 != parent2] <-
                stats::rbinom(sum(parent1 - parent2 != 0), 1,
                       prob = parent1r / (parent1r + parent2r))
            child2[parent1 != parent2] <-
```

```
                stats::rbinom(sum(parent1 - parent2 != 0), 1,
                       prob = parent2r / (parent1r + parent2r))
        }
        return(rbind(as.integer(child1), as.integer(child2)))
    } else {
        child1 <- parent1
        child2 <- parent2
        return(rbind(child1, child2))
    }
}

# This function defines the ways of mutation
# the mutation will be implemented based on
# the user-defined probability or the default
# if the user input an invalid probability (not between 0 and 1)
# we generate an error message
# which is 1/P*(C)^0.5

# mutation_rate is an optional numeric value between 0
# and 1 indicating mutation rate
# child is a vector containing the chromosome of child produced by crossover
# P is an integer indicating the number of parent chromosomes
# C is an integer indicating the number of predictor variables or genes

mutate_child <- function(mutation_rate, child, P, C) {

    if (is.null(mutation_rate)) {
        return(as.integer(abs(round(child, 0) -
                        stats::rbinom(C, 1,
                                    prob = 1 / (P * sqrt(C))))))
    } else {

        if (mutation_rate > 1 | mutation_rate < 0) {
            stop("Error: mutation_rate must be between 0 and 1")
        }

        return(as.integer(abs(round(child, 0) -
                        stats::rbinom(C, 1,
                                    prob = mutation_rate))))
    }
}
```

We then create each of the P individuals in the next generation through the function "create_next_generation". We ensure that each child/new model should have at least one 1 in their genes/design variables, and the output of the function "create_next_generation" is the next generation "generation_t1".

```
# This function will create children through different methods of
# selection, crossover and mutation
# a future generation will be of the same size of the previous generation
# the output is the child generation

# the inputs are:
# obj_fun_output is a numeric vector containg the objective function output
# for each chromosome for ranking
```

```r
# select_parents, crossover_parents, and mutate_child
# are sub functions defined above
# pCrossover is the probability of crossover between the parents
# the default is 0.8
# mutation_rate is the probability of mutation between parents
# the default is 1/P*(C)^0.5

create_next_generation <- function(generation_t0, obj_fun_output,
                                   select_parents,
                                   crossover_method,
                                   crossover_parents,
                                   pCrossover,
                                   mutate_child,
                                   mutation_rate) {

    # set variables
    P <- dim(generation_t0)[1]
    C <- dim(generation_t0)[2]
    parent_rank <- obj_fun_output[, 2]

    #Create matrix for next generation
    generation_t1 <- matrix(NA, dim(generation_t0)[1], dim(generation_t0)[2])

    #########
    #Selection, Crossover, and Mutation
    #########

    i <- 1 #initialize while loop
    while(i <= dim(generation_t1)[1]) {

        # Selection ----------------
        parentInd <- select_parents(parent_rank)

        # Crossover ----------------
        children <- crossover_parents(generation_t0, parentInd,
                                      crossover_method, pCrossover, parent_rank)

        # Mutation ----------------
        child1 <- mutate_child(mutation_rate, children[1, ], P, C)
        child2 <- mutate_child(mutation_rate, children[2, ], P, C)

        # Check if all zeros -----------------
        if (!all(child1 == 0)  & !all(child2 == 0)) {
          generation_t1[c(i, i + 1), ] <- rbind(child1, child2)

            # update counter
            i <- i + 2
        }
    }

    # return new new generation
    return(generation_t1)
}
```

Step 4: Looping and Termination

We at last implementing the genetic algorithm to a dataset to select the most relevant predictors (X) for the response variable (Y) through using our primary function – "select". We first check whether each input's type is reasonable, then we implement the genetic algorithm in the following steps: 1. we generate the initial population using function "generate_founders". 2. we evaluate the fitness of each model in the initial population using function "evaluate_fitness" and select the optimal parent pairs. 3. we create a child generation through different selection, crossover, and mutation methods. 4. we repeat steps 2 and 3 until either the maximum allowed number of iterations is reached or the result converges to a specific optimum. Note that we define the criteria for convergence as the top 25% in the population having imperceptible difference in their fitness values. Also, users can input their desired maximum number of iterations or use the default of 100.

```
##############
#primary function for package
##############

# This function will implement the steps above and continue to
# generate future generations until criteria is met
# it will also check the reasonableness of the type of each input
# the output is the best model selected, its fitness value,
# the number of iterations the algorithm takes, whether the
# algorithm converges, and the time each iteration takes

# the inputs are:
# Y is a vector of response variable
# X is a matrix or dataframe of predictor variables
# family is a character string describing the error distribution
# and link function to be used in the model, the default is gaussian
# objective_function is a function for computing objective
# the default is AIC, user can specify customized function
# crossover_parents_function is a function for crossover between mate pairs
# user can specify customized function
# crossover_method is a character string describing crossover method
# the default is a three-point crossover
# pCrossover is a numeric value for he probability of crossover for each mate pair
# start_chrom a numeric value for the  size of the popuation of chromosomes
# mutation_rate a numeric value for rate of mutation
# converge a logical value indicating whether algorithm should attempt to
# converge or run for specified number of iterations
# tol a numeric value indicating convergence tolerance
# the default is 1e-4
# iter an integer specifying maximum number of generations algorithm will produce
# the default is 100

select <- function(Y, X, family = "gaussian",
                   objective_function = stats::AIC,
                   crossover_parents_function = crossover_parents,
                   crossover_method = c("method1", "method2", "method3"),
                   pCrossover = 0.8,
                   start_chrom = NULL,
                   mutation_rate = NULL,
                   converge = TRUE,
                   tol = 1e-4,
                 iter = 100,
                 minimize = TRUE,
                 nCores = 1L) {

    ########
    #error checking
    ########

    # X
    if (!is.matrix(X) & !is.data.frame(X)) stop("X must be matrix or dataframe")

    # Y
    if (!is.vector(Y) & !is.matrix(Y)) stop("Y must be vector or 1 column matrix")
    if (is.matrix(Y)) {
        if(ncol(Y) > 1) stop("Y must be vector or 1 column matrix")
    }

    # family
    if (family == "gaussian" & all(Y %% 1 == 0)) {cat("Warning: outcome distribution is are 1, 0 intege
    if (family == "gamma" & sum(Y > 0) > 0) {cat("Warning: outcome values < 0, family == 'gamma' may pr

    # objective_function
    if (!is.function(objective_function)) stop("Error: objective_function must be a function")

    # crossover_parents
    if (!is.function(crossover_parents_function)) stop("Error: crossover_parents must be a function")

    # crossover_method
    if (!is.character(crossover_method)) stop("Error: crossover_method should be a character string")
    if (length(crossover_method) > 1) crossover_method <- crossover_method[1]

    # pCrossover
    if (!is.numeric(pCrossover) | pCrossover < .Machine$double.eps | pCrossover > 1) stop("Error: pCros
    if (pCrossover < 0.5) cat("Warning: pCrossover < 0.5 may not reach optimum")

    # mutation_rate
    if (!is.null(mutation_rate)) {
        if(!is.numeric(mutation_rate)) stop("Error: mutation rate must be numeric")
        if(mutation_rate < 0 | mutation_rate > 1) stop("Error: mutation rate must be bewteen 0 and 1")
    }

    # converge
    if (!is.logical(converge)) stop("Error: converge must be logical (TRUE/FALSE). Default is TRUE")

    # tol
    if (!is.numeric(tol)) stop("Error: tol must be numeric. Default is 1e-4")

    # iter
    if (!is.numeric(iter)) stop("Error: iter must be numeric")
    if (length(iter) > 1) stop("Error: iter be of length one")

    # minimize
    if (!is.logical(minimize)) stop("Error: minimize must be logical (TRUE/FALSE). Default is TRUE")
```

```r
# nCores
if (!is.integer(nCores)) stop("Error: nCores must be integer of length 1")
if (nCores > parallel::detectCores()) stop("Error: nCores cannot be larger than detectCores()")
if (nCores < 1) stop("Error: nCores must be >= 1")

##########
# Perform genetic algorithm
#########
t1 <- Sys.time() # timing

# Step 1: Generate founders ----------------
generation_t0 <- generate_founders(X, start_chrom)
P <- nrow(generation_t0) #num chromosomes
cat("1. Generate founders: ", P, "chromosomes")

# Step 2. Evaluate founder fitness Fitness of inital pop ----------------
cat("\n2. Evaluate founders")
obj_fun_output_t0 <- evaluate_fitness(generation_t0, Y, X,
                                      family,
                                      nCores, minimize,
                                      objective_function,
                                      rank_objective_function)

#create array to store fitness data for each iteration
convergeData <- array(dim = c(P, 2, 1)) #P x 2 x iter

#save founder fitness evaluation data
convergeData[, , 1] <- obj_fun_output_t0[
                        order(obj_fun_output_t0[, 2], decreasing = T),
                        c(1, 3)]

# Step 3. loop through successive generations  ----------------
cat("\n3. Begin breeding \n Generations: ")
t1 <- c(t1, Sys.time())
for (i in 1:iter) {

    # 1. create next generation ----------------
    if (i == 1) {
        generation_t1 <- generation_t0
        obj_fun_output_t1 <- obj_fun_output_t0
    }

    generation_t1 <- create_next_generation(generation_t1,
                                            obj_fun_output_t1,
                                            select_parents,
                                            crossover_method,
                                            crossover_parents_function,
                                            pCrossover,
                                            mutate_child,
                                            mutation_rate)

    # 2. evaluate children fitness ----------------
    obj_fun_output_t1 <- evaluate_fitness(generation_t1, Y, X,
                                            family,
                                            nCores, minimize,
                                            objective_function,
                                            rank_objective_function)

    # store fitness data
    convergeData <- abind::abind(convergeData,
                                 obj_fun_output_t1[order(obj_fun_output_t1[, 2],
                                 decreasing = T), c(1, 3)])

    # cat generation and save timing
    cat(i, "-", sep = "")

    # 3. check convergence ----------------
    if (i > 10 & isTRUE(converge)) {
        if(isTRUE(all.equal(mean(convergeData[1:(P * 0.25), 2, i]),
                                 convergeData[1, 2, i],
                                 check.names = F,
                                 tolerance = tol)) &
           isTRUE(all.equal(mean(convergeData[1:(P * 0.25), 2, (i - 1)]),
                                 convergeData[1, 2, i],
                                 check.names = F,
                                 tolerance = tol))) {

        #if((abs(convergeData[1, 2, i] - convergeData[1, 2, (i - 1)]) /
        #          abs( convergeData[1, 2, (i - 1)])) < tol) {
            cat("\n#### Converged! ####")
            break
        }
    }
}

# Step 4. process output ----------------
t1 <- c(t1, Sys.time())

# get models with the best fitness
best_scores <- convergeData[, , i]
if (sum(best_scores[, 2] == best_scores[1, 2]) > 1) {
    num_best_scores <- sum(best_scores[, 2] == best_scores[1, 2])
} else {num_best_scores <- 1}
bestModel <- generation_t1[convergeData[, 1, i], ]

# other output information
value <- convergeData[1, 2, dim(convergeData)[3]]
if(dim(convergeData)[3] < iter) {
  converged <- "Yes"
} else {
  converged <- "No"
}

# create output list
output <- list("Best_model" =
                   colnames(X)[round(colMeans(bestModel[1:dim(best_scores)[1], ]), 0) == 1],
                optimize = list("obj_func" = paste(substitute(objective_function))[3],
                                value = as.numeric(round(value, 4)),
                                minimize = minimize,
                                method = crossover_method),
                iter = dim(convergeData)[3],
                converged = converged,
               convergeData = convergeData,
               timing = t1)

# set class
class(output) <- c("GA", class(output))

return(output)
}
```

In addition to the functions above, we also define a function named “plot.GA”, which will plot the values of fitness for each model in a generation, the mean fitness of each generation, as well as the best fitness obtained in each generation.

```
# This function will plot the output of the function select

# the inputs are:
# x is a GA object and it must be a class of GA
# col1 is a character string for color of chromosomes scatter plotpoints
# the default is blue
# col2 is a character string for color of chromosomes mean and max lines
# the default is red

plot.GA <- function(x, ..., col1 = "blue", col2 = "red") {

  # error checking
  if (!is.character(col1)) stop("Error: col1 must be a character string of length 1")
  if (length(col1) > 1) stop("Error: col1 must be a character string of length 1")
  if (!is.character(col2)) stop("Error: col1 must be a character string of length 1")
  if (length(col2) > 1) stop("Error: col1 must be a character string of length 1")

  # get plot data from GA object
  convergeData <- x$convergeData
  obj_fun <- as.character(x$optimize[1])
  minimize <- x$optimize[3]
  method <- as.character(x$optimize[4])
  iter <- x$iter

  # scatter plot of chromosome data across generations ----------------

  # check for scales packages
  if (requireNamespace("scales", quietly = TRUE)) {

    # with scales
    graphics::plot(jitter(rep(1, nrow(convergeData))), convergeData[, 2, 1], type = "p",
                   pch = 19, col = scales::alpha(col1, 0.1),
                   ylim = c(min(convergeData[ , 2, ], na.rm = T),
                            max(convergeData[ , 2, ], na.rm = T)),
                   xlim = c(1, iter), xlab = "Generations", ylab = obj_fun,
                    main = paste("GA performance: \n ", obj_fun, method))
    for (i in 2:iter) {
      graphics::points(jitter(rep(i, nrow(convergeData))),
                       convergeData[, 2, i], type = "p",
                       pch = 19, col = scales::alpha(col1, 0.25))
    }
  } else {

    # without scales
    graphics::plot(jitter(rep(1, nrow(convergeData))), convergeData[, 2, 1], type = "p",
                   pch = 19, col = col1,
                   ylim = c(min(convergeData[ , 2, ], na.rm = T),
                            max(convergeData[ , 2, ], na.rm = T)),
                   xlim = c(1, iter), xlab = "Generations", ylab = obj_fun,
                   main = paste("GA performance: \n ", obj_fun, method))
    for (i in 2:iter) {
      graphics::points(jitter(rep(i, nrow(convergeData))),
                       convergeData[, 2, i], type = "p",
                       pch = 19, col = "blue")
    }
  }

  # line plots of mean and max fitness per generation
  graphics::lines(1:iter, sapply(1:iter,
                                 function(x) convergeData[1, 2, x]), type = "l",
                  col = col2, lwd = 2)
  graphics::lines(1:iter, sapply(1:iter,
                                 function(x) mean(convergeData[, 2, x])),
                  col = col2, lty = 2, lwd = 2)

  # legend
  if (minimize == TRUE) {
    location <- "topright"
  } else {
    location <- "bottomright"
  }
  graphics::legend(location, c("Chr fitness", "Best fitness", "Mean Fitness"),
                   col = c(col1, col2, col2), pch = c(19, NA, NA), lwd = c(NA, 2, 2),
                   lty = c(NA, 1, 2), bty = "n")
}
```

II. Testings Performed

We then perform formal testing for each of the function in our algorithm.

```
library(testthat)
library(MASS)
```

We use "mtcars" to test the function "generate_founders", where we test two cases: the user defines the number of chromsomes/models in a population or not. For each case, we test whether the output from "generate_founders" is a $P * C$ matrix, and whether each row of the output contains only numerical values of 0 and 1.

```r
y <- mtcars$mpg
x <- as.matrix(mtcars[,c(-1)])
C <- dim(x)[2]
P <- 2 * C
t0 <- generate_founders(x,12)

# test the dimension, class, and type of the output
test_that('generate_founders works',
          {test <- generate_founders(x, NULL)
            expect_equal(dim(test)[2], C)
            expect_equal(dim(test)[1], P)
            expect_true(is.matrix(test))
            expect_true(is.numeric(test))
            expect_false(any(rowSums(test == 0) == C))})

test_that('generate_founders works with user defined chrom_start',
          {test <- generate_founders(x, 25)
            expect_equal(dim(test)[2], C)
            expect_equal(dim(test)[1], 25)
            expect_true(is.matrix(test))
            expect_true(is.numeric(test))
expect_false(any(rowSums(test == 0) == C))})
```

We test the function "rank_objective_function" with simulated data, and consider when the user desires a maximization or a minimization of the targeted fitness function. In each case, we consider the dimension, class, and type of the output.

```r
objfunout <- runif(100, 0, 1)
rnk1 <- rank_objective_function(objfunout, T)

# Test the dimension, class, and type of the output of rank_objective_function
# with minimize set to TRUE.
test_that('test rank_objective_function works',{
  expect_equal(dim(rnk1), c(100,3))
  expect_true(is.numeric(rnk1))
  expect_true(is.matrix(rnk1))
})

rnk2 <- rank_objective_function(objfunout, F)

# Test the dimension, class, and type of the output of rank_objective_function
# with minimize set to FALSE.
test_that('test rank_objective_function works',{
  expect_equal(dim(rnk2), c(100,3))
  expect_true(is.numeric(rnk2))
  expect_true(is.matrix(rnk2))
})
```

We test the function "evaluate_fitness" with the dataset "mtcars". We first test whether the parallel processing works and check the type of the function output. While the default of the objective function is AIC, we check the functionality of this function by inputting some other possible objective function, such as the log-likehood function.

```r
# test data ----------------
Y <- as.matrix(mtcars$mpg)
```

```r
X <- as.matrix(mtcars[2:ncol(mtcars)])

# get input data
C <- dim(X)[2] # number genes
P <- 2 * C # number of chromosomes

# generate chromosomes to test
geneSample <- sample(c(0, 1),
                     replace = TRUE,
                     size = ceiling(1.2 * C * P))

indices <- seq_along(geneSample)
firstGen <- split(geneSample, ceiling(indices / C))
generation_t0 <- matrix(unlist(unique(firstGen)[1:P]),
                        ncol = C, byrow = TRUE)
generation_t0 <- generation_t0[apply(generation_t0, 1,
                                     function(x) !all(x == 0)), ]

# serial evaluation
test_that('serial fitness evaluation works',
          {test <- evaluate_fitness(generation_t0, Y, X,
                                    family = "gaussian",
                                    nCores = 1, minimize = TRUE,
                                    objective_function = stats::AIC,
                                    rank_objective_function)
          expect_is(test, "matrix")
          expect_type(test, "double")
})

# parallel evaluation
test_that('parallel fitness evaluation works',
          {test <- evaluate_fitness(generation_t0, Y, X,
                                   family = "gaussian",
                                   nCores = 2, minimize = TRUE,
                                   objective_function = stats::AIC,
                                   rank_objective_function)
          expect_is(test, "matrix")
          expect_type(test, "double")
})

# test maximize evaluation and other objective functions
test_that('Other objective_functions work',
          {test <- evaluate_fitness(generation_t0, Y, X,
                                   family = "gaussian",
                                   nCores = 1, minimize = FALSE,
                                   objective_function = stats::logLik,
                                   rank_objective_function)
          expect_is(test, "matrix")
          expect_type(test, "double")
})
```

We then test the functions implemented in the selection, crossover, and mutation step using the dataset "mtcars". For the function "select_parents", we check the type, length, and values containing in the output (at least one of the design variable should be 1). For the function "crossover_parents", we implementing the formal testing for each of the three methods, where we consider the class, dimension, and the values involved in the output (we should only have 0's and 1's in the output, and the output should have at least one 1). We test the function "mutate_child" using random binomial data, and we consider three cases: the mutation happens under the default mutation rate, a valid mutation rate (a probability between 0 and 1), and an invalid rate. We test the class, dimension, and the values involved in the output in each case.

```r
#test data
Y <- as.matrix(mtcars$mpg)
X <- as.matrix(mtcars[2:ncol(mtcars)])
dim(X)
```

```
## [1] 32 10
```

```r
# get input data
C <- dim(X)[2] # number genes
P <- 2 * C # number of chromosomes

# generate chromosomes to test
geneSample <- sample(c(0, 1),
                     replace = TRUE,
                     size = ceiling(1.2 * C * P))

indices <- seq_along(geneSample)
firstGen <- split(geneSample, ceiling(indices / C))
generation_t0 <- matrix(unlist(unique(firstGen)[1:P]),
                        ncol = C, byrow = TRUE)
generation_t0 <- generation_t0[apply(generation_t0, 1,
                                     function(x) !all(x == 0)), ]

parentInd <- sample(1:P, 2, replace = F)
parent_rank <- 1:P

# select_parents ----------------
test_that("select_parents works ",
          {test <- select_parents(parent_rank)
              expect_is(test, "integer")
              expect_type(test, "integer")
              expect_true(all(!is.na(test))) # not all 0's
              expect_equal(length(test), 2) # 2 parents
              expect_true(all(test > 0 & test <= P)) # C chromosomes
})


# crossover_parents ----------------
test_that("crossover_parents works and crossover method 1 works",
          {test <- crossover_parents(generation_t0, parentInd,
                        crossover_method = "method1", pCrossover = 1, parent_rank)
          expect_is(test, "matrix")
          expect_type(test, "integer")
          expect_true(any(test == 1 | test == 0)) # 0's and 1's
          expect_true(!all(test == 0)) # not all 0's
          expect_equal(dim(test)[1], 2) # 2 children
          expect_equal(dim(test)[2], C) # C chromosomes
          })
```

```r
test_that("crossover_parents works and crossover method 2 works",
          {test <- crossover_parents(generation_t0, parentInd,
                        crossover_method = "method2", pCrossover = 1, parent_rank)
            expect_is(test, "matrix")
            expect_type(test, "integer")
            expect_true(any(test == 1 | test == 0)) # 0's and 1's
            expect_true(!all(test == 0)) # not all 0's
            expect_equal(dim(test)[1], 2) # 2 children
            expect_equal(dim(test)[2], C) # C chromosomes
            })

test_that("crossover_parents works and crossover method 3 works",
          {test <- crossover_parents(generation_t0, parentInd,
                                     crossover_method = "method3", pCrossover = 1, parent_rank)
            expect_is(test, "matrix")
            expect_type(test, "integer")
            expect_true(any(test == 1 | test == 0)) # 0's and 1's
            expect_true(!all(test == 0)) # not all 0's
            expect_equal(dim(test)[1], 2) # 2 children
            expect_equal(dim(test)[2], C) # C chromosomes
            })

# mutate_child ----------------
mutation_rate <- NULL
child <- stats::rbinom(C, 1, runif(1, min = 0.35, max = 0.65))

test_that("mutate_child works when mutation_rate is null",
          {test <- mutate_child(mutation_rate, child, P, C)
            expect_type(test, "integer")
            expect_true(any(test == 1 | test == 0)) # 0's and 1's
            expect_true(!all(test == 0)) # not all 0's
            expect_equal(length(test), C) # C chromosomes
          })

test_that("mutate_child works when user specifies a mutation rate",
          {test <- mutate_child(mutation_rate = 0.01, child, P, C)
          expect_type(test, "integer")
          expect_true(any(test == 1 | test == 0)) # 0's and 1's
          expect_true(!all(test == 0)) # not all 0's
          expect_equal(length(test), C) # C chromosomes
          })

test_that("mutate_child works when user specifies a mutation rate",
          {test <- mutate_child(mutation_rate = 0.01, child, P, C)
          expect_type(test, "integer")
          expect_true(any(test == 1 | test == 0)) # 0's and 1's
          expect_true(!all(test == 0)) # not all 0's
          expect_equal(length(test), C) # C chromosomes
          })

test_that("mutate_child breaks when user specifies an incorrect mutation rate",
            {expect_error(mutate_child(mutation_rate = 1.2, child, P, C))
            expect_error(mutate_child(mutation_rate = -0.2, child, P, C))
})
```

We test the function “create_next_generation” with the dataset “mtcars”. We first test its functionality when the first corssover method is applied. For valid inputs of the probabilities for crossover and mutation, we check the dimension, class and type of the function, and for invalid inputs of these probabilities, we checked if the function will throw an error message. We also test if “create_next_generation” functions correctly when the second and third crossover methods are implemented.

```
Y <- as.matrix(mtcars$mpg)
X <- as.matrix(mtcars[2:ncol(mtcars)])
dim(X)
```

```
## [1] 32 10
```

```
# get input data
C <- dim(X)[2] # number genes
P <- 2* C # number of chromosomes

# generate chromosomes to test
geneSample <- sample(c(0, 1),
                     replace = TRUE,
                     size = ceiling(1.2 * C * P))

indices <- seq_along(geneSample)
firstGen <- split(geneSample, ceiling(indices / C))
generation_t0 <- matrix(unlist(unique(firstGen)[1:P]),
                        ncol = C, byrow = TRUE)
generation_t0 <- generation_t0[apply(generation_t0, 1,
                                     function(x) !all(x == 0)), ]


dim(generation_t0)
```

```
## [1] 20 10
```

```
obj_fun_output <- evaluate_fitness(generation_t0, Y, X,
                                   family = "gaussian",
                                   nCores = 1, minimize = TRUE,
                                   objective_function = stats::AIC,
                                   rank_objective_function)



crossover_method<-"method1"
# basic evaluation
test_that('create next generation works',
          {
          #Assign a probability in crossovber
          pCrossover<-1

          #Assign mutation rate
          mutation_rate<-0.1
          test <- create_next_generation(generation_t0, obj_fun_output,
                                         select_parents,
                                         crossover_method,
                                         crossover_parents,
                                         pCrossover,
                                                    mutate_child,
                                                    mutation_rate)
          expect_equal(dim(test), c(20,10))
          expect_is(test, "matrix")
          expect_type(test, "integer")
          })

# another one with different probability
test_that('other probability works',
          {
          #Assign a probability in crossovber
          pCrossover<-0

          #Assign mutation rate
          mutation_rate<-1
          test <- create_next_generation(generation_t0, obj_fun_output,
                                         select_parents,
                                         crossover_method,
                                         crossover_parents,
                                         pCrossover,
                                         mutate_child,
                                         mutation_rate)
          expect_equal(dim(test), c(20,10))
          expect_is(test, "matrix")
          expect_type(test, "integer")
          })

# test whether invalid input will throw error
test_that('serial fitness evaluation works',

          {
          #Assign a probability in crossovber
          pCrossover<-1.1

          #Assign mutation rate
          mutation_rate<-1
          expect_error(test <- create_next_generation(generation_t0, obj_fun_output,
                                                    select_parents,
                                                    crossover_method,
                                                    crossover_parents,
                                                    pCrossover,
                                                    mutate_child,
                                                    mutation_rate))
          })

test_that('serial fitness evaluation works',

          {
            #Assign a probability in crossovber
            pCrossover<--0.1

            #Assign mutation rate
            mutation_rate<-1
            expect_error(test <- create_next_generation(generation_t0, obj_fun_output,
                                                        select_parents,
                                                        crossover_method,
                                                        crossover_parents,
                                                        pCrossover,
                                                        mutate_child,
                                                        mutation_rate))
          })

test_that('serial fitness evaluation works',

          {
            #Assign a probability in crossovber
            pCrossover<-1

            #Assign mutation rate
            mutation_rate<--1
            expect_error(test <- create_next_generation(generation_t0, obj_fun_output,
                                                        select_parents,
                                                        crossover_method,
                                                        crossover_parents,
                                                        pCrossover,
                                                        mutate_child,
                                                        mutation_rate))
          })

test_that('serial fitness evaluation works',

          {
            #Assign a probability in crossovber
            pCrossover<-1

            #Assign mutation rate
            mutation_rate<-1.3
            expect_error(test <- create_next_generation(generation_t0, obj_fun_output,
                                                        select_parents,
                                                        crossover_method,
                                                        crossover_parents,
                                                        pCrossover,
                                                        mutate_child,
                                                        mutation_rate))
          })

#Make sure other method works
test_that('methods 2 works',

          {
            #Assign a probability in crossovber
            pCrossover<-1
            #Assign Different methods
            crossover_method<-"method2"
            #Assign mutation rate
            mutation_rate<-1
            test <- create_next_generation(generation_t0, obj_fun_output,
                                               select_parents,
                                               crossover_method,
                                               crossover_parents,
                                               pCrossover,
                                               mutate_child,
                                               mutation_rate)
            expect_equal(dim(test), c(20,10))
            expect_is(test, "matrix")
            expect_type(test, "integer")
          })

test_that('methods 3 works',

          {
            #Assign a probability in crossovber
            pCrossover<-1

            #Assign mutation rate
            mutation_rate<-1

            #Assign Method
            crossover_method<-"method3"
            test <- create_next_generation(generation_t0, obj_fun_output,
                                               select_parents,
                                               crossover_method,
                                               crossover_parents,
                                               pCrossover,
                                               mutate_child,
                                               mutation_rate)
            expect_equal(dim(test), c(20,10))
            expect_is(test, "matrix")
            expect_type(test, "integer")
})
```

At last, we test our primary function "select" with the "mtcars" data, where we check the closeness between the optimal fitness value obtained from our genetic algorithm with the value of this fitness function in linear regression. We test each of the three crossover methods under an either the default function of AIC or an alternative function such as BIC, and the set the tolerance of the difference to be 1 in this case.

```
fit <- lm(mpg~., data = mtcars)
step <- stepAIC(fit, direction="both")
```

```
## Start:  AIC=70.9
## mpg ~ cyl + disp + hp + drat + wt + qsec + vs + am + gear + carb
##
##        Df Sum of Sq    RSS    AIC
## - cyl   1    0.0799 147.57 68.915
## - vs    1    0.1601 147.66 68.932
## - carb  1    0.4067 147.90 68.986
## - gear  1    1.3531 148.85 69.190
## - drat  1    1.6270 149.12 69.249
## - disp  1    3.9167 151.41 69.736
## - hp    1    6.8399 154.33 70.348
## - qsec  1    8.8641 156.36 70.765
## <none>              147.49 70.898
## - am    1   10.5467 158.04 71.108
## - wt    1   27.0144 174.51 74.280
##
## Step:  AIC=68.92
## mpg ~ disp + hp + drat + wt + qsec + vs + am + gear + carb
##
##        Df Sum of Sq    RSS    AIC
## - vs    1    0.2685 147.84 66.973
## - carb  1    0.5201 148.09 67.028
## - gear  1    1.8211 149.40 67.308
## - drat  1    1.9826 149.56 67.342
## - disp  1    3.9009 151.47 67.750
## - hp    1    7.3632 154.94 68.473
## <none>              147.57 68.915
## - qsec  1   10.0933 157.67 69.032
## - am    1   11.8359 159.41 69.384
## + cyl   1    0.0799 147.49 70.898
## - wt    1   27.0280 174.60 72.297
##
## Step:  AIC=66.97
## mpg ~ disp + hp + drat + wt + qsec + am + gear + carb
##
##        Df Sum of Sq    RSS    AIC
## - carb  1    0.6855 148.53 65.121
## - gear  1    2.1437 149.99 65.434
## - drat  1    2.2139 150.06 65.449
## - disp  1    3.6467 151.49 65.753
## - hp    1    7.1060 154.95 66.475
## <none>              147.84 66.973
## - am    1   11.5694 159.41 67.384
## - qsec  1   15.6830 163.53 68.200
## + vs    1    0.2685 147.57 68.915
## + cyl   1    0.1883 147.66 68.932
## - wt    1   27.3799 175.22 70.410
##
## Step:  AIC=65.12
## mpg ~ disp + hp + drat + wt + qsec + am + gear
##
##        Df Sum of Sq    RSS    AIC
## - gear  1     1.565 150.09 63.457
## - drat  1     1.932 150.46 63.535
## <none>              148.53 65.121
## - disp  1    10.110 158.64 65.229
## - am    1    12.323 160.85 65.672
## - hp    1    14.826 163.35 66.166
## + carb  1     0.685 147.84 66.973
## + vs    1     0.434 148.09 67.028
## + cyl   1     0.414 148.11 67.032
## - qsec  1    26.408 174.94 68.358
## - wt    1    69.127 217.66 75.350
##
## Step:  AIC=63.46
## mpg ~ disp + hp + drat + wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## - drat  1     3.345 153.44 62.162
## - disp  1     8.545 158.64 63.229
## <none>              150.09 63.457
## - hp    1    13.285 163.38 64.171
## + gear  1     1.565 148.53 65.121
## + cyl   1     1.003 149.09 65.242
## + vs    1     0.645 149.45 65.319
## + carb  1     0.107 149.99 65.434
## - am    1    20.036 170.13 65.466
## - qsec  1    25.574 175.67 66.491
## - wt    1    67.572 217.66 73.351
##
## Step:  AIC=62.16
## mpg ~ disp + hp + wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## - disp  1     6.629 160.07 61.515
## <none>              153.44 62.162
## - hp    1    12.572 166.01 62.682
## + drat  1     3.345 150.09 63.457
## + gear  1     2.977 150.46 63.535
## + cyl   1     2.447 150.99 63.648
## + vs    1     1.121 152.32 63.927
## + carb  1     0.011 153.43 64.160
## - qsec  1    26.470 179.91 65.255
## - am    1    32.198 185.63 66.258
## - wt    1    69.043 222.48 72.051
##
## Step:  AIC=61.52
## mpg ~ hp + wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## - hp    1     9.219 169.29 61.307
## <none>              160.07 61.515
## + disp  1     6.629 153.44 62.162
## + carb  1     3.227 156.84 62.864
## + drat  1     1.428 158.64 63.229
## - qsec  1    20.225 180.29 63.323
## + cyl   1     0.249 159.82 63.465
## + vs    1     0.249 159.82 63.466
## + gear  1     0.171 159.90 63.481
## - am    1    25.993 186.06 64.331
## - wt    1    78.494 238.56 72.284
##
## Step:  AIC=61.31
## mpg ~ wt + qsec + am
##
##        Df Sum of Sq    RSS    AIC
## <none>              169.29 61.307
## + hp    1     9.219 160.07 61.515
## + carb  1     8.036 161.25 61.751
## + disp  1     3.276 166.01 62.682
## + cyl   1     1.501 167.78 63.022
## + drat  1     1.400 167.89 63.042
## + gear  1     0.123 169.16 63.284
## + vs    1     0.000 169.29 63.307
## - am    1    26.178 195.46 63.908
## - qsec  1   109.034 278.32 75.217
## - wt    1   183.347 352.63 82.790
```

```
step
```

```
##
## Call:
## lm(formula = mpg ~ wt + qsec + am, data = mtcars)
##
## Coefficients:
## (Intercept)           wt         qsec           am
##       9.618       -3.917        1.226        2.936
```

```
## test for the closeness between AIC
# test for method1
test_that('Variable selection using genetic algorithms',{
  test1 <- select(Y, X, family = "gaussian",
                  crossover_method = 'method1')

  expect_equal(test1$optimize$value, AIC(lm(mpg~wt+qsec+am, data = mtcars)),
               tolerance = 1e0)
})
```

```
## 1. Generate founders:  20 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31-32
## #### Converged! ####
```

```
# test for method2
test_that('Variable selection using genetic algorithms',{
  test2 <- select(Y, X, family = "gaussian",
                  crossover_method = 'method2')

  expect_equal(test2$optimize$value, AIC(lm(mpg~ wt+qsec+am, data = mtcars)),
               tolerance = 1e0)
})
```

```
## 1. Generate founders:  20 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-
## #### Converged! ####
```

```
# test for method3
test_that('Variable selection using genetic algorithms',{
  test3 <- select(Y, X, family = "gaussian",
                  crossover_method = 'method3')

  expect_equal(test3$optimize$value, AIC(lm(mpg~wt+qsec+am, data = mtcars)),
               tolerance = 1e0)
```

```r
})
```

```
## 1. Generate founders:  20 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-
## #### Converged! ####
```

```r
## test for the closeness between BIC
# test for method1
test_that('Variable selection using genetic algorithms',{
  test1 <- select(Y, X, family = "gaussian",
                  crossover_method = 'method1')

  expect_equal(test1$optimize$value, BIC(lm(mpg~wt+qsec+am, data = mtcars)),
               tolerance = 1e0)
})
```

```
## 1. Generate founders:  20 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-
## #### Converged! ####
```

```r
# test for method2
test_that('Variable selection using genetic algorithms',{
  test1 <- select(Y, X, family = "gaussian",
                  crossover_method = 'method2')

  expect_equal(test1$optimize$value, BIC(lm(mpg~wt+qsec+am, data = mtcars)),
               tolerance = 1e0)
})
```

```
## 1. Generate founders:  20 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-
## #### Converged! ####
```

```r
# test for method3
test_that('Variable selection using genetic algorithms',{
  test1 <- select(Y, X, family = "gaussian",
                  crossover_method = 'method3')

  expect_equal(test1$optimize$value, BIC(lm(mpg~wt+qsec+am, data = mtcars)),
               tolerance = 1e0)
})
```

```
## 1. Generate founders:  20 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-
## #### Converged! ####
```

From the output, we can see that each function in the algorithm passes the test successfully.

III. Real Implementations

In addition to testing our algorithm on the “mtcars” dataset, we also implement our algorithm on a larger dataset–“Ionosphere”, which has 30+ possible predictors in this case. The variables “V1” and “V2” are two nominal and the variable “Class” is factor, we ignore these three variables, and fit all other variables on “V34”. Again, we compare the optimal fitness value obtained from our genetic algorithm with that in a linear regression model (note that this linear model is selected to be the optimal model obtained from the stepwise selection by AIC). We also plot the results from the genetic algorithm for each of the three methods using the function “GA.plot”.

```
library('mlbench')
data("Ionosphere")
Ionosphere$V1<-NULL
Ionosphere$V2<-NULL
Ionosphere$Class<-NULL

y<-Ionosphere[,32]
x<- as.matrix(Ionosphere[,-32])

library(testthat)
library(MASS)
fit <- lm(V34 ~ ., data = Ionosphere)
step <- stepAIC(fit, direction="both")
```

```
## Start:  AIC=-784.31
## V34 ~ V3 + V4 + V5 + V6 + V7 + V8 + V9 + V10 + V11 + V12 + V13 +
##     V14 + V15 + V16 + V17 + V18 + V19 + V20 + V21 + V22 + V23 +
##     V24 + V25 + V26 + V27 + V28 + V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V10   1    0.0011 31.311 -786.30
## - V22   1    0.0107 31.321 -786.19
## - V21   1    0.0127 31.323 -786.17
## - V17   1    0.0133 31.324 -786.16
## - V3    1    0.0163 31.327 -786.13
## - V6    1    0.0242 31.334 -786.04
## - V26   1    0.0252 31.335 -786.03
## - V24   1    0.0368 31.347 -785.90
## - V28   1    0.1021 31.412 -785.17
## <none>              31.310 -784.31
## - V25   1    0.1913 31.502 -784.17
## - V19   1    0.2937 31.604 -783.03
## - V12   1    0.3025 31.613 -782.94
## - V7    1    0.3344 31.645 -782.58
## - V27   1    0.3394 31.650 -782.53
## - V23   1    0.3620 31.672 -782.28
## - V20   1    0.5554 31.866 -780.14
## - V14   1    0.6422 31.952 -779.18
## - V29   1    0.7340 32.044 -778.18
## - V4    1    0.7883 32.099 -777.58
## - V18   1    0.9413 32.252 -775.91
## - V31   1    0.9899 32.300 -775.39
## - V11   1    1.0069 32.317 -775.20
## - V9    1    1.2678 32.578 -772.38
## - V33   1    1.3462 32.656 -771.53
## - V16   1    1.3707 32.681 -771.27
## - V13   1    1.4877 32.798 -770.02
## - V30   1    1.6553 32.966 -768.23
## - V15   1    2.0890 33.399 -763.64
## - V5    1    2.6226 33.933 -758.08
## - V8    1    2.9311 34.241 -754.90
## - V32   1    4.2657 35.576 -741.48
##
## Step:  AIC=-786.3
## V34 ~ V3 + V4 + V5 + V6 + V7 + V8 + V9 + V11 + V12 + V13 + V14 +
##     V15 + V16 + V17 + V18 + V19 + V20 + V21 + V22 + V23 + V24 +
##     V25 + V26 + V27 + V28 + V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V22   1    0.0105 31.322 -788.18
## - V17   1    0.0126 31.324 -788.16
## - V21   1    0.0134 31.325 -788.15
## - V3    1    0.0164 31.328 -788.11
## - V6    1    0.0243 31.336 -788.03
## - V26   1    0.0250 31.336 -788.02
## - V24   1    0.0358 31.347 -787.90
## - V28   1    0.1011 31.412 -787.17
## <none>               31.311 -786.30
## - V25   1    0.1946 31.506 -786.12
## - V19   1    0.3008 31.612 -784.94
## - V12   1    0.3198 31.631 -784.73
## - V7    1    0.3355 31.647 -784.56
## - V27   1    0.3400 31.651 -784.51
## + V10   1    0.0011 31.310 -784.31
## - V23   1    0.3609 31.672 -784.28
## - V20   1    0.5589 31.870 -782.09
## - V14   1    0.6428 31.954 -781.17
## - V29   1    0.7450 32.056 -780.05
## - V4    1    0.7874 32.099 -779.58
## - V18   1    0.9450 32.256 -777.86
## - V31   1    0.9895 32.301 -777.38
## - V11   1    1.0173 32.329 -777.08
## - V9    1    1.2674 32.579 -774.37
## - V33   1    1.3488 32.660 -773.50
## - V16   1    1.3754 32.687 -773.21
## - V13   1    1.5308 32.842 -771.54
## - V30   1    1.6577 32.969 -770.19
## - V15   1    2.0879 33.399 -765.64
## - V5    1    2.6217 33.933 -760.08
## - V8    1    2.9327 34.244 -756.87
## - V32   1    4.2817 35.593 -743.31
##
## Step:  AIC=-788.18
## V34 ~ V3 + V4 + V5 + V6 + V7 + V8 + V9 + V11 + V12 + V13 + V14 +
##     V15 + V16 + V17 + V18 + V19 + V20 + V21 + V23 + V24 + V25 +
##     V26 + V27 + V28 + V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V17   1    0.0114 31.333 -790.05
## - V3    1    0.0159 31.338 -790.00
## - V6    1    0.0222 31.344 -789.93
## - V26   1    0.0254 31.347 -789.90
## - V24   1    0.0277 31.349 -789.87
## - V21   1    0.0283 31.350 -789.86
## - V28   1    0.0971 31.419 -789.10
## <none>               31.322 -788.18
## - V25   1    0.1966 31.518 -787.99
## - V19   1    0.2903 31.612 -786.94
## - V12   1    0.3166 31.638 -786.65
## - V7    1    0.3251 31.647 -786.56
## + V22   1    0.0105 31.311 -786.30
## - V27   1    0.3557 31.677 -786.22
## + V10   1    0.0008 31.321 -786.19
## - V23   1    0.3592 31.681 -786.18
## - V20   1    0.5771 31.899 -783.77
## - V14   1    0.6327 31.954 -783.16
## - V29   1    0.7422 32.064 -781.96
## - V4    1    0.7784 32.100 -781.56
## - V18   1    0.9467 32.268 -779.73
## - V31   1    0.9795 32.301 -779.37
## - V11   1    1.0379 32.360 -778.74
## - V9    1    1.2587 32.581 -776.35
## - V33   1    1.3423 32.664 -775.45
## - V16   1    1.3651 32.687 -775.21
## - V13   1    1.5292 32.851 -773.45
## - V30   1    1.6497 32.972 -772.16
## - V15   1    2.0790 33.401 -767.62
## - V5    1    2.6767 33.998 -761.40
## - V8    1    3.4680 34.790 -753.32
## - V32   1    4.3536 35.675 -744.50
##
## Step:  AIC=-790.05
## V34 ~ V3 + V4 + V5 + V6 + V7 + V8 + V9 + V11 + V12 + V13 + V14 +
##     V15 + V16 + V18 + V19 + V20 + V21 + V23 + V24 + V25 + V26 +
##     V27 + V28 + V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V3    1    0.0168 31.350 -791.86
## - V6    1    0.0230 31.356 -791.80
## - V26   1    0.0230 31.356 -791.80
## - V24   1    0.0254 31.359 -791.77
## - V21   1    0.0356 31.369 -791.66
## - V28   1    0.0984 31.432 -790.95
## <none>               31.333 -790.05
## - V25   1    0.2503 31.584 -789.26
## - V19   1    0.2859 31.619 -788.87
## - V12   1    0.3187 31.652 -788.50
## - V7    1    0.3250 31.658 -788.43
## + V17   1    0.0114 31.322 -788.18
## + V22   1    0.0093 31.324 -788.16
## + V10   1    0.0002 31.333 -788.06
## - V23   1    0.3708 31.704 -787.92
## - V27   1    0.3930 31.726 -787.68
## - V20   1    0.5763 31.910 -785.66
## - V14   1    0.6332 31.966 -785.03
## - V29   1    0.7308 32.064 -783.96
## - V4    1    0.7808 32.114 -783.41
## - V18   1    0.9361 32.269 -781.72
## - V31   1    0.9977 32.331 -781.05
## - V11   1    1.1523 32.486 -779.38
## - V9    1    1.3182 32.651 -777.59
## - V16   1    1.4224 32.756 -776.47
## - V33   1    1.4542 32.787 -776.13
## - V30   1    1.6516 32.985 -774.02
## - V13   1    1.7159 33.049 -773.34
## - V15   1    2.0978 33.431 -769.31
## - V5    1    2.8575 34.191 -761.42
## - V8    1    3.5566 34.890 -754.32
## - V32   1    4.3443 35.678 -746.48
##
## Step:  AIC=-791.86
## V34 ~ V4 + V5 + V6 + V7 + V8 + V9 + V11 + V12 + V13 + V14 + V15 +
##     V16 + V18 + V19 + V20 + V21 + V23 + V24 + V25 + V26 + V27 +
##     V28 + V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V26   1    0.0217 31.372 -793.62
## - V6    1    0.0243 31.374 -793.59
## - V24   1    0.0336 31.384 -793.49
## - V21   1    0.0483 31.398 -793.32
## - V28   1    0.1064 31.456 -792.68
## <none>              31.350 -791.86
## - V25   1    0.2345 31.585 -791.25
## - V19   1    0.3087 31.659 -790.43
## - V7    1    0.3104 31.660 -790.41
## + V3    1    0.0168 31.333 -790.05
## + V17   1    0.0123 31.338 -790.00
## + V22   1    0.0088 31.341 -789.96
## - V23   1    0.3541 31.704 -789.92
## + V10   1    0.0003 31.350 -789.87
## - V12   1    0.3880 31.738 -789.55
## - V27   1    0.4072 31.757 -789.34
## - V20   1    0.5604 31.910 -787.65
## - V14   1    0.6301 31.980 -786.88
## - V29   1    0.7153 32.065 -785.95
## - V4    1    0.8597 32.210 -784.37
## - V18   1    0.9760 32.326 -783.10
## - V31   1    0.9884 32.338 -782.97
## - V11   1    1.1507 32.501 -781.21
## - V9    1    1.3805 32.731 -778.74
## - V33   1    1.4409 32.791 -778.09
## - V16   1    1.4414 32.791 -778.09
## - V30   1    1.6534 33.003 -775.83
## - V13   1    1.7400 33.090 -774.90
## - V15   1    2.1032 33.453 -771.07
## - V5    1    2.9266 34.277 -762.54
## - V8    1    3.5583 34.908 -756.13
## - V32   1    4.3442 35.694 -748.31
##
## Step:  AIC=-793.62
## V34 ~ V4 + V5 + V6 + V7 + V8 + V9 + V11 + V12 + V13 + V14 + V15 +
##     V16 + V18 + V19 + V20 + V21 + V23 + V24 + V25 + V27 + V28 +
##     V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V6    1    0.0288 31.401 -795.30
## - V24   1    0.0501 31.422 -795.06
## - V21   1    0.0606 31.432 -794.95
## - V28   1    0.1332 31.505 -794.13
## <none>              31.372 -793.62
## - V25   1    0.2237 31.595 -793.13
## - V7    1    0.3037 31.675 -792.24
## - V19   1    0.3090 31.681 -792.18
## + V26   1    0.0217 31.350 -791.86
## + V3    1    0.0154 31.356 -791.80
## + V17   1    0.0099 31.362 -791.73
## + V22   1    0.0093 31.362 -791.73
## + V10   1    0.0002 31.371 -791.63
## - V23   1    0.3599 31.732 -791.62
## - V12   1    0.4092 31.781 -791.07
## - V27   1    0.4149 31.787 -791.01
## - V20   1    0.5868 31.959 -789.12
## - V14   1    0.6524 32.024 -788.40
## - V29   1    0.7200 32.092 -787.66
## - V18   1    0.9582 32.330 -785.06
## - V31   1    0.9687 32.340 -784.95
## - V4    1    0.9842 32.356 -784.78
## - V11   1    1.1838 32.556 -782.62
## - V9    1    1.3589 32.731 -780.74
## - V33   1    1.4486 32.820 -779.78
## - V16   1    1.5617 32.933 -778.57
## - V30   1    1.6913 33.063 -777.19
## - V13   1    1.7888 33.161 -776.16
## - V15   1    2.0926 33.464 -772.96
## - V5    1    2.9932 34.365 -763.64
## - V8    1    3.5779 34.950 -757.71
## - V32   1    4.3880 35.760 -749.67
##
## Step:  AIC=-795.3
## V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 + V14 + V15 +
##     V16 + V18 + V19 + V20 + V21 + V23 + V24 + V25 + V27 + V28 +
##     V29 + V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V24   1    0.0400 31.441 -796.85
## - V21   1    0.0765 31.477 -796.45
## - V28   1    0.1188 31.519 -795.98
## <none>              31.401 -795.30
## - V25   1    0.2057 31.606 -795.01
## - V19   1    0.3080 31.709 -793.87
## - V7    1    0.3279 31.728 -793.65
## + V6    1    0.0288 31.372 -793.62
## + V26   1    0.0262 31.374 -793.59
## + V3    1    0.0166 31.384 -793.49
## - V23   1    0.3477 31.748 -793.44
## + V17   1    0.0106 31.390 -793.42
## + V22   1    0.0071 31.393 -793.38
## + V10   1    0.0003 31.400 -793.30
## - V12   1    0.3985 31.799 -792.87
## - V27   1    0.4120 31.813 -792.72
## - V20   1    0.5802 31.981 -790.87
## - V14   1    0.6769 32.077 -789.81
## - V29   1    0.7856 32.186 -788.63
## - V18   1    0.9371 32.338 -786.98
## - V31   1    0.9418 32.342 -786.93
## - V4    1    1.1450 32.546 -784.73
## - V11   1    1.2868 32.687 -783.20
## - V9    1    1.4057 32.806 -781.93
## - V33   1    1.4321 32.833 -781.65
## - V16   1    1.6051 33.006 -779.80
## - V30   1    1.6723 33.073 -779.09
## - V13   1    1.7678 33.168 -778.08
## - V15   1    2.3270 33.728 -772.21
## - V5    1    2.9865 34.387 -765.41
## - V8    1    3.6822 35.083 -758.38
## - V32   1    5.2757 36.676 -742.79
##
## Step:  AIC=-796.85
## V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 + V14 + V15 +
##     V16 + V18 + V19 + V20 + V21 + V23 + V25 + V27 + V28 + V29 +
##     V30 + V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V21   1    0.0680 31.508 -798.10
## - V28   1    0.1614 31.602 -797.06
## <none>              31.441 -796.85
## - V25   1    0.1885 31.629 -796.76
## - V19   1    0.2966 31.737 -795.56
## + V26   1    0.0409 31.400 -795.31
## + V24   1    0.0400 31.401 -795.30
## - V23   1    0.3222 31.763 -795.28
## + V3    1    0.0253 31.415 -795.14
## - V7    1    0.3381 31.779 -795.10
## + V6    1    0.0187 31.422 -795.06
## + V17   1    0.0072 31.433 -794.93
## + V22   1    0.0005 31.440 -794.86
## + V10   1    0.0001 31.440 -794.85
## - V12   1    0.3911 31.832 -794.51
## - V27   1    0.4413 31.882 -793.96
## - V20   1    0.5528 31.993 -792.74
## - V14   1    0.7373 32.178 -790.72
## - V29   1    0.7577 32.198 -790.49
## - V18   1    0.9343 32.375 -788.58
## - V31   1    0.9510 32.392 -788.39
## - V4    1    1.1064 32.547 -786.71
## - V11   1    1.2608 32.701 -785.05
## - V9    1    1.4151 32.856 -783.40
## - V33   1    1.5321 32.973 -782.15
## - V16   1    1.6175 33.058 -781.25
## - V13   1    1.7443 33.185 -779.90
## - V30   1    1.8194 33.260 -779.11
## - V15   1    2.3594 33.800 -773.45
## - V5    1    2.9824 34.423 -767.04
## - V8    1    3.6674 35.108 -760.13
## - V32   1    5.2719 36.712 -744.44
##
## Step:  AIC=-798.1
## V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 + V14 + V15 +
##     V16 + V18 + V19 + V20 + V23 + V25 + V27 + V28 + V29 + V30 +
##     V31 + V32 + V33
##
##        Df Sum of Sq    RSS     AIC
## - V28   1    0.1379 31.646 -798.56
## <none>               31.508 -798.10
## - V25   1    0.2361 31.745 -797.48
## - V19   1    0.2752 31.784 -797.04
## + V21   1    0.0680 31.441 -796.85
## + V26   1    0.0573 31.451 -796.73
## + V3    1    0.0426 31.466 -796.57
## + V6    1    0.0321 31.476 -796.45
## + V24   1    0.0314 31.477 -796.45
## + V17   1    0.0169 31.492 -796.28
## + V22   1    0.0149 31.494 -796.26
## + V10   1    0.0001 31.508 -796.10
## - V12   1    0.3679 31.876 -796.02
## - V23   1    0.3901 31.899 -795.78
## - V27   1    0.3921 31.901 -795.76
## - V7    1    0.4608 31.969 -795.00
## - V20   1    0.5177 32.026 -794.38
## - V14   1    0.6951 32.204 -792.44
## - V29   1    0.7680 32.276 -791.64
## - V31   1    0.9302 32.439 -789.88
## - V18   1    1.0224 32.531 -788.89
## - V4    1    1.2360 32.745 -786.59
## - V11   1    1.2884 32.797 -786.03
## - V9    1    1.4419 32.950 -784.39
## - V33   1    1.4680 32.976 -784.11
## - V16   1    1.5918 33.100 -782.80
## - V13   1    1.9431 33.452 -779.09
## - V30   1    2.0822 33.591 -777.63
## - V15   1    2.2921 33.801 -775.45
## - V5    1    3.2881 34.797 -765.25
## - V8    1    3.6009 35.109 -762.11
## - V32   1    5.7903 37.299 -740.88
##
## Step:  AIC=-798.56
## V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 + V14 + V15 +
##     V16 + V18 + V19 + V20 + V23 + V25 + V27 + V29 + V30 + V31 +
##     V32 + V33
```

```
##
##        Df Sum of Sq    RSS     AIC
## <none>              31.646 -798.56
## + V28   1    0.1379 31.508 -798.10
## - V19   1    0.2409 31.887 -797.90
## + V26   1    0.1011 31.545 -797.69
## + V24   1    0.0685 31.578 -797.32
## + V3    1    0.0571 31.589 -797.20
## - V25   1    0.3074 31.954 -797.17
## + V21   1    0.0444 31.602 -797.06
## - V12   1    0.3435 31.990 -796.77
## + V17   1    0.0133 31.633 -796.71
## + V6    1    0.0105 31.636 -796.68
## + V22   1    0.0033 31.643 -796.60
## + V10   1    0.0023 31.644 -796.59
## - V27   1    0.3881 32.034 -796.28
## - V23   1    0.4515 32.098 -795.59
## - V7    1    0.4773 32.124 -795.31
## - V20   1    0.5264 32.173 -794.77
## - V14   1    0.7433 32.390 -792.41
## - V29   1    0.7469 32.393 -792.38
## - V18   1    0.9220 32.568 -790.48
## - V31   1    1.0067 32.653 -789.57
## - V4    1    1.2272 32.874 -787.21
## - V11   1    1.2873 32.934 -786.57
## - V9    1    1.4284 33.075 -785.07
## - V16   1    1.7976 33.444 -781.17
## - V33   1    1.8880 33.534 -780.22
## - V13   1    2.0635 33.710 -778.39
## - V30   1    2.2529 33.899 -776.43
## - V15   1    2.3555 34.002 -775.36
## - V5    1    3.2466 34.893 -766.28
## - V8    1    3.5684 35.215 -763.06
## - V32   1    7.3755 39.022 -727.03
```

```
step
```

```
##
## Call:
## lm(formula = V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 +
##     V14 + V15 + V16 + V18 + V19 + V20 + V23 + V25 + V27 + V29 +
##     V30 + V31 + V32 + V33, data = Ionosphere)
##
## Coefficients:
## (Intercept)           V4           V5           V7           V8
##    -0.04433     -0.17495     -0.31081      0.12630      0.25277
##          V9          V11          V12          V13          V14
##     0.23697     -0.18497     -0.08356      0.26041      0.12383
##         V15          V16          V18          V19          V20
##    -0.28863      0.20513     -0.14054     -0.08103      0.11829
##         V23          V25          V27          V29          V30
##     0.11646      0.07873      0.09590      0.12846      0.21667
##         V31          V32          V33
##     0.15362      0.35333     -0.24896
```

```
# the optimal model selected by stepwise AIC is
# V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 +
# V14 + V15 + V16 + V18 + V19 + V20 + V23 + V25 + V27 + V29 +
# V30 + V31 + V32 + V33

AIC(lm( V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 +
          V14 + V15 + V16 + V18 + V19 + V20 + V23 + V25 + V27 + V29 +
          V30 + V31 + V32 + V33, data = Ionosphere))
```

```
## [1] 199.5313
```

```
test_that('Variable selection using genetic algorithms',{
  test1 <-GA::select(y, x,
                  crossover_method = 'method1', nCores = 1L)

  expect_equal(test1$optimize$value, AIC(lm( V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 +
                                             V14 + V15 + V16 + V18 + V19 + V20 + V23 + V25 +
               tolerance = 2e0)
})
```

```
## 1. Generate founders:  62 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31-32
## #### Converged! ####
```

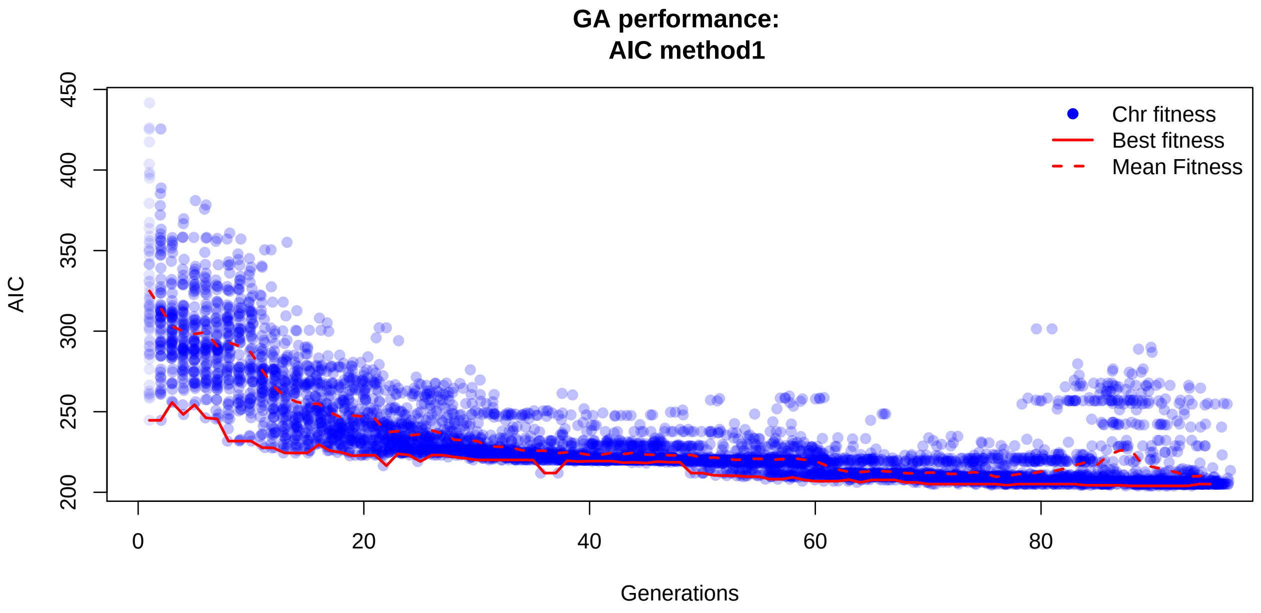
```
test1 <-GA::select(y, x, crossover_method = 'method1', nCores = 1L)
```

```
## 1. Generate founders:  62 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31-32
## #### Converged! ####
```

```
test1$optimize$value
```

```
## [1] 203.9655
```

```
plot(test1, col1 = "blue", col2 = "red")
```



```
test_that('Variable selection using genetic algorithms',{
  test2 <-GA::select(y, x,
                     crossover_method = 'method2', nCores = 1L)

  expect_equal(test2$optimize$value, AIC(lm( V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 +
                                     V14 + V15 + V16 + V18 + V19 + V20 + V23 + V25 +
              tolerance = 2e0)
})
```

```
## 1. Generate founders:  62 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-
## #### Converged! ####
```

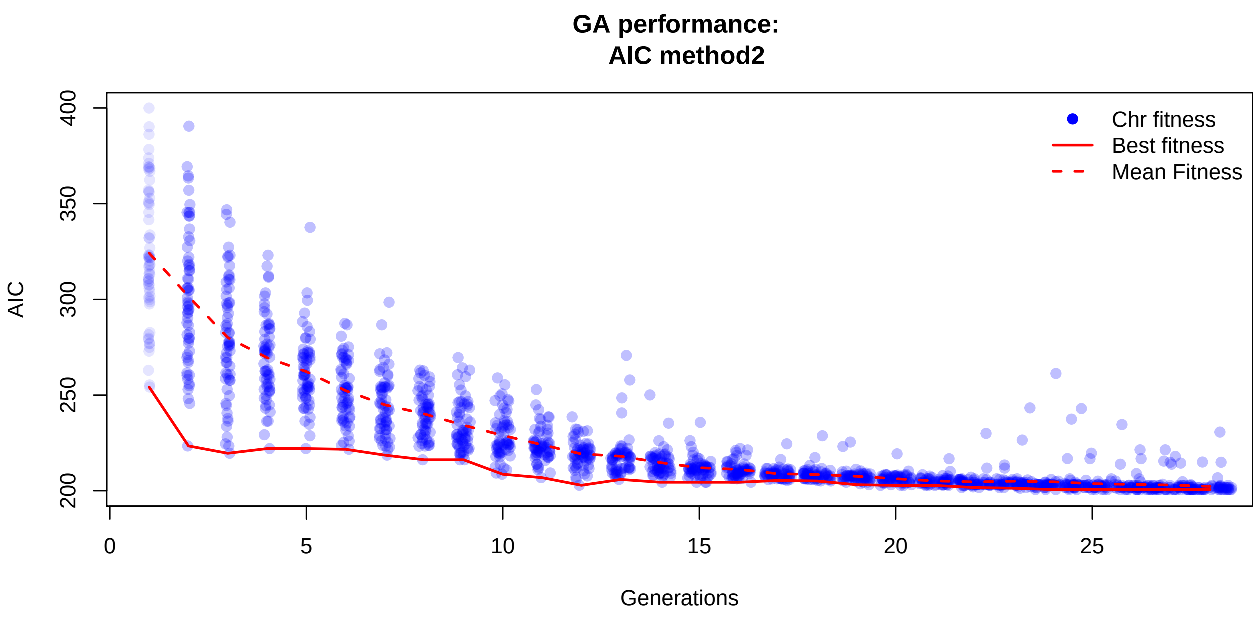
```
test2 <-GA::select(y, x, crossover_method = 'method2', nCores = 1L)
```

```
## 1. Generate founders:  62 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-
## #### Converged! ####
```

```
test2$optimize$value
```

```
## [1] 199.999
```

```
plot(test2, col1 = "blue", col2 = "red")
```



```
test_that('Variable selection using genetic algorithms',{
  test1 <-GA::select(y, x,
                      crossover_method = 'method3', nCores = 1L)

  expect_equal(test1$optimize$value, AIC(lm( V34 ~ V4 + V5 + V7 + V8 + V9 + V11 + V12 + V13 +
                                              V14 + V15 + V16 + V18 + V19 + V20 + V23 + V25 +
              tolerance = 2e0)
})
```

```
## 1. Generate founders:  62 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31-32-
## #### Converged! ####
```

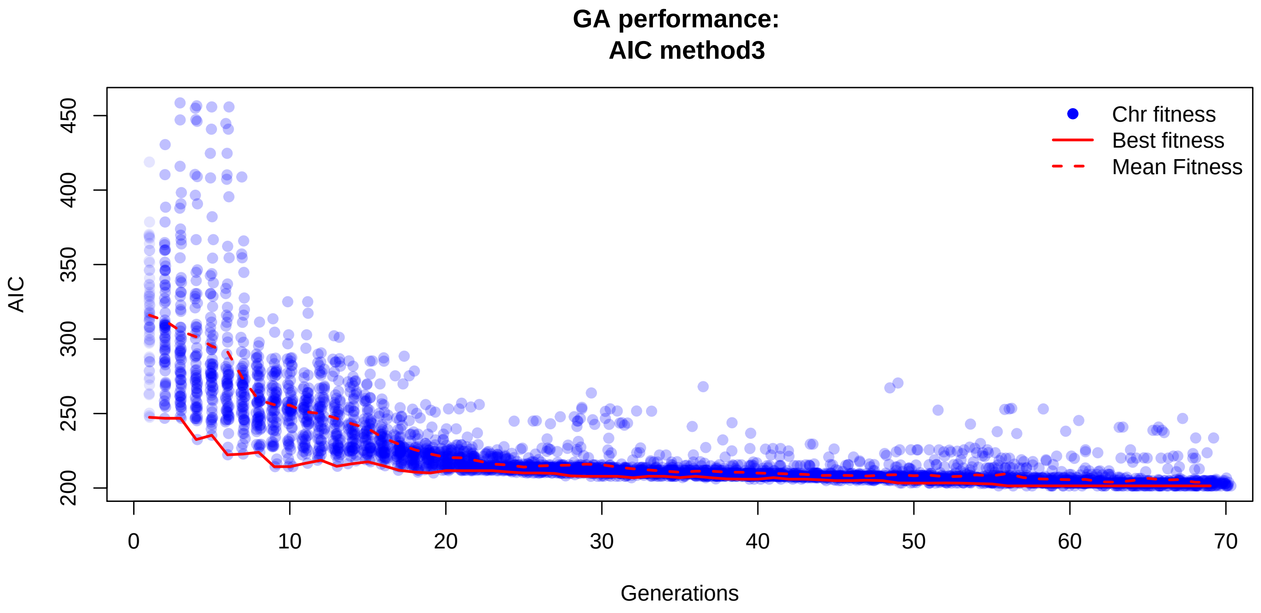
```
test3 <-GA::select(y, x, crossover_method = 'method3', nCores = 1L)
```

```
## 1. Generate founders:  62 chromosomes
## 2. Evaluate founders
## 3. Begin breeding
##  Generations: 1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31-32-
## #### Converged! ####
```

```
test3$optimize$value
```

```
## [1] 201.4152
```

```
plot(test3, col1 = "blue", col2 = "red")
```



We can see that all three crossover methods in the algorithm pass the formal testing. From the plots, we can see that the three-point crossover method converges more slowly than the other methods, and it still has relatively large variation in the objective function (AIC in this case) among different chomosomes/models. The second crossover method seems to perform the best in terms of the speed of convergence, and it also has smaller variance in the fitness function for the latter generations. The third crossover method seems to perform better than the three-point crossover method, while it has slower convergence and more variation compared to the second method.

IV. Members' Contributions

In this project, all members in the group help test the practicability of each function and propose constructive suggestions in designing the algorithm, especially on the set-up of the crossover and mutation methods, as well as the criteria of convergence. Then each group member is also responsible for the following major parts:

Cameron Adams: wrote and continuously improve the functionality of functions including "select", developed the third crossover method, wrote the help information "select.Rd" and the formal testing for functions such as "select_parents".

Yuwen Chen: wrote the formal testing for functions including "generate_founders" and "select", constructed the GA package and ensure that it is operational, debugged some impracticabilities in the algorithm.

Weijie Xu: constructed the basic structure of the algorithm and wrote functions including "generate_founders", developed the second crossover method , wrote the formal testing for functions such as "create_next_generation".

Yilin Zhou: checked the functionality of each step and the improve the consistensy of the presentation, constructed and wrote this documentation, double check to ensure the project attains all the requirements.

VI. References

Geof H. Givens, Jennifer A. Hoeting (2013) Combinatorial Optimization (italicize). Chapter 3 of Computational Statistics (italicize).

Henderson and Velleman (1981), Building multiple regression models interatively. Biometrics, 37, 391-411.

Sigillito, V. G., Wing, S. P., Hutton, L. V., Baker, K. B. (1989). Classification of radar returns from the ionosphere using neural networks. Johns Hopkins APL Technical Digest, 10, 262-266.