Stat243 Final Group Project

Cameron Adams, Wendi 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 crossover and possible mutation (with function "create_next_generation")
4. Looping and termination: repeat steps 2 and 3 until either the maxmimum 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 that ranges from C and 2C is recommended for binary codings of chromosomes or predictive models (where C is the length of the chromosome or the number of predictors), we calculate P as 2C. Also, since P is between 10 and 200 in most real applications, we set 10 and 200 as the lower and upper bounds of P in this case. We then randomly sample 1.2*C*P 0's and 1's, splitting them up into the P individuals or 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.

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

#initiative founding chromosomes
generate_founders <- function(X) {

    # number of predictors ---------------
    C <- dim(X)[2]

    # number of founders ----------------
    P <- 2 * C
    if (P > 200) {P <- 200
    } else if (P < 10) {
        P <- 10}

    #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], 1,
                                    function(x) !all(x == 0)), ]

    return(generation_t0)
}
```

Step 2: Evaluation

We then evaluate the fitness of each chromosome or model using the function "evaluate_fitness". We get the values of variables such as "family" and "objFun" from their parent frames, which are our primary function–"select" in this case. We consider several variations depending on the users' inputs: whether our variable selection is based on linear regression or GLM, what we should use for our fitness function (AIC, BIC, log-likelihood or a user-defined function), and whether the user requires parallel processing. We calculate the value of the fitness function for each model, and rank these values. Note that we use negative AIC or negative BIC to rank a model if it uses AIC or BIC as its fitness function, while we rank the models using their log-likehood if log-likehood is the objective criterion. We return the rank and the value of the fitness function as our output for each model.

```
# evaluate the generation using lm/glm and the objective function
# default is AIC

evaluate_fitness <- function(generation_t0, Y, X) {

    # get objects from their parent frames----------------
    family <- get("family", mode = "any", envir = parent.frame())
    objFun <- get("objFun", mode = "any", envir= parent.frame())
    parallel <- get("parallel", mode = "any", envir= parent.frame())
    minimize <- get("minimize", mode = "any", envir= parent.frame())

    #test parms
    #family <- "gaussian"
    ##objFun <- "AIC"
    #minimize <- TRUE
    #parallel <- FALSE

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

    # calculate objective function ----------------
    calc_objective_function <- function(mod) {

        objFun <- get("objFun", mode = "any", envir = parent.frame())
        if (objFun == "AIC") {
            extractAIC(mod)[2]
        } else if (objFun == "BIC") {
            BIC(mod)
        } else if (objFun == "logLik") {
            logLik(mod)
        } else if (objFun == "user") {
            user_func(mod)
        }
    }

    # rank obj function output ----------------
    rank_objective_function <- function(objFunOutput){

        objFun <- get("objFun", mode = "any", envir = parent.frame())
        minimize <- get("minimize", mode = "any", envir = parent.frame())

        if (objFun %in% c("AIC", "BIC") | minimize == TRUE) {
            r <- rank(-objFunOutput, na.last = TRUE, ties.method = "first")
        } else if (objFun == "logLike" | minimize == FALSE) {
            r <- rank(objFunOutput, na.last = TRUE, ties.method = "first")
        }
        return(cbind(chr = 1:P, rank = r, objFunOutput))
    }

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

    # serial ----------------
    if (parallel == FALSE) {

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

    # parallel ----------------
    } else if (parallel == TRUE) {

        # mclapply options ----------------
        nCores <- detectCores() - 1
        if(dim(X)[1] < 1000) {
          preschedule <- FALSE
        } else {
          preschedule <- TRUE
        }

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

    # rank ----------------
    rank <- rank_objective_function(objFunOutput)

    #return rankings ----------------
    return(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 function “create_next_generation”. Since the population size of a future generation remains the same as the initial generation, we would still have P individuals or models in the next generation. We calculate the probability of selection for a parent as $\phi_i = \frac{2*r_i}{P*(P+1)}$. For each individual or model in this next generation, we sample two parents from the previous generation, where the first parent is sampled by the selection probability and the second parent is selected at random.

We then implement three ways of crossover and mutation to generate two children from the parents.

First method: uses two-point crossover, where we choose two non-overlapping points within the length of the chromosome or the number of the design variables in each model. We swap the elements between the two points, rendering two children. We then perform mutation with a mutation rate of $\frac{1}{P\times\sqrt{C}}$ (each gene or design variable has a probability of $\frac{1}{P\times\sqrt{C}}$ to flip).

Second method:

Through implementing selection, crossover and mutation for each individual or model in the next generation, we obtain our next generation of “generation_t1”.

```
create_next_generation <- function(generation_t0, objFunOutput_t0, iter) {
    #will create children, selecting mate pairs based upon
    #with higher objectFun scores
    #nextGen will be of same size of prev generation
    #will then perform mutation on resulting generation

    #phi: prob of selection, softmax using objFun output
    #P: # of chromosomes/models
```

```r
    #C: # of genes/vars
    #r: object function
    #nextGeneration: stores new generation
    #P1, P2: selected parents

    #inherit variables
    crossMeth <- get("crossMeth", mode = "any", envir = parent.frame())

    #set variables
    P <- dim(generation_t0)[1]
    C <- dim(generation_t0)[2]
    rank <- objFunOutput_t0[, 2]
    score <- objFunOutput_t0[, 3]

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

    #set up plot to view sampling in each generation
    #require(scales)
    #plot(phi, score)
    #legend("bottomleft", c("parents", "sampled parents"), pch = c(21, 19))

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

    #keep high rank parents and put in new generation
    #goodGeneration_t0 <- generation_t0[rank > quantile(rank, probs = 0.95), ]
    #generation_t1[1:dim(goodGeneration_t0)[1], ] <- goodGeneration_t0

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

    i <- 1 #dim(goodGeneration_t0)[1] + 1 #initialize while loop
    while(i <= dim(generation_t1)[1]) {

        #SELECTION: select Parents
        #method 1: both parents selected by rank
            #parentInd <- sample(1:P, 2, prob=phi, replace = F) #this is better than method 2

        #method 2: first parent by rank, second random
            parentInd <- c(sample(1:P, 1, prob=phi, replace = F),
                           sample(1:P, 1, replace = F))

            parents <- generation_t0[parentInd, ]
            parentRank <- rank[parentInd]
            parentScore <- score[parentInd]

        #update plot with sampled parents
        #points(phi[parentInd], score[parentInd], pch = 19,
        #       col = alpha("black", 0.5))

        #CROSSOVER and MUTATION parent to  create child
        if (crossMeth == "method1") {

            #METHOD 1:
                #crossover: two crossover points, non-overlapping
                #creates two children
                cross1 <- sample(2:(C/2-1), 1)
                cross2 <- sample((C/2+1):(C-1), 1)
                child1 <- c(parents[1, 1:cross1],
                            parents[2, (cross1+1):cross2],
                            parents[1, (cross2+1):C])

                child2 <- c(parents[1, 1:cross1],
                            parents[2, (cross1+1):cross2],
                            parents[1, (cross2+1):C])

                #mutation:
                child1 <- abs(round(child1, 0) -
                                rbinom(C, 1, prob = 1 / (P * sqrt(C))))
                child2 <- abs(round(child2, 0) -
                                 rbinom(C, 1, prob = 1 / (P * sqrt(C))))


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

            #METHOD 2:
                #crossover: method upweights parent with higher rank high
                #create one child
                childProb <- parents[1, ] * parentRank[1] /
                        (parentRank[1] + parentRank[2]) +
                                parents[2, ] * parentRank[2]  /
                        (parentRank[1] + parentRank[2])
                #mutation: this method has A LOT of mutation
                child1 <- rbinom(C, 1, prob = childProb)
                child2 <- rbinom(C, 1, prob = childProb)

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

            #METHOD 3
                #randomly samples non-concordant variables
                #between parents, slightly upweights parent selected
                #by prob. proportional to rank
                #this provide mutation as well
                #2 children
                child1 <- parents[1, ]
                child2 <- parents[2, ]
                child1[parents[1, ] != parents[2, ]] <-
                    rbinom(sum(parents[1, ] - parents[2, ] != 0), 1,
                        prob = parentRank[1] / (parentRank[1] + parentRank[2]))
                child2[parents[1, ] != parents[2, ]] <-
                    rbinom(sum(parents[1, ] - parents[2, ] != 0), 1,
                        prob = 1 - (parentRank[1] / (parentRank[1] + parentRank[2])))
        }
        #check if duplicate child, and for all non-zero genes
        #if (all(!rowSums(generation_t1 == child1 |
        #                 generation_t1 == child2, na.rm = T) == C) &
        #    sum(child1) > 0 & sum(child2) > 0) {
            #not dup: add child to new generation
            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 steps above to datasets to select the most relevant predictors (X) for the response variable (Y) through using our primary function – "select". Note that the defaults of the number of iterations, the objective function, and the family are 100, AIC, and gaussian (linear regression) respectively. We first generate the initial population, based on which we continue to create future generations until either the maximum allowed number of iterations is reached or the result converge to a specific optimum.

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

select <- function(Y, X, iter, objFun = c("AIC", "BIC", "logLik", "user"),
                  family = "gaussian",
                  crossMeth = c("method1", "method2", "method3"),
                  numChromosomes = NULL,
                  pCrossover = 1,
                  user_objFun = NULL,
                  converge = TRUE, minimize = TRUE, parallel = FALSE) {

    #input objects
    if(missing(iter)) {
      iter <- 100
      }
    if(missing(objFun)) {
      objFun <- "AIC"
      }
    if(missing(family)) {
      family <- "gaussian"
      }
    if(missing(converge)) {
      converge <- FALSE
      }
    if(missing(minimize)) {
      minimize <- TRUE
      }
    if(missing(parallel)) {
      parallel <- FALSE
      }
```

```
  if(missing(user_objFun)) {
    user_objFun <- NULL
    }
  require(doParallel)
  require(abind)

  #error checking

  #function objects

  # print settings

  ##########
  # Perform genetic algorithm
  #########

  #step 1: Generate founders -----------------
      generation_t0 <- generate_founders(X)
      P <- nrow(generation_t0) #num chromosomes
      cat("1. Generate founders: ", P, "chromosomes")
      t1 <- Sys.time()

  #Step 2. Evaluate founder fitness Fitness of inital pop  -----------------
      objFunOutput_t0 <- evaluate_fitness(generation_t0, Y, X)
      cat("\n2. Evaluate founder fitness")
      t1 <- c(t1, Sys.time())

      #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] <- objFunOutput_t0[
                              order(objFunOutput_t0[, 2], decreasing = T),
                              c(1, 3)]

  #Step 3. loop through successive generations until either:
      #1. finish default or user specified number of iterations
      #2. convergence to specific optimum
      tol = sqrt(.Machine$double.eps) #tolerance for converage check
      converged <- 0

      cat("\n3. Begin breeding \n Generations: ")

      for (i in 1:iter) {

              #breeding: create children
              if (i == 1) {
                  generation_t1 <- generation_t0
                  objFunOutput_t1 <- objFunOutput_t0
              }

              generation_t1 <- create_next_generation(generation_t1,
                                                      objFunOutput_t1, iter)
```

```
                # eval fitness ----------------
                objFunOutput_t1 <- evaluate_fitness(generation_t1, Y, X)

                # store fitness data ----------------
                convergeData <- abind(convergeData,
                    objFunOutput_t1[order(objFunOutput_t1[, 2],
                                            decreasing = T), c(1, 3)])

                # cat generation and save timing ----------------
                cat(i,"-", sep = "")
                t1 <- c(t1, Sys.time())

                # check convergence ----------------
                if (i > 10 & converge == TRUE) {
                    if(isTRUE(all.equal(mean(convergeData[1:(P * 0.5), 2, i]),
                                        convergeData[1, 2, i],
                                        check.names = F,
                                        tolerance = tol))) {
                        converged <- converged + 1
                        if (converged >= 1) {
                        cat("\n#### Converged! ####")
                        break
                        }
                    }
                }
            }

    #process output ----------------
    bestModel <- generation_t1[convergeData[1, 1, i], ]
    value <- convergeData[1, 2, dim(convergeData)[3]]
    if(dim(convergeData)[3] < iter) {converged <- "Yes"
    } else {converged <- "No"}

    output <- list("BestModel" = bestModel,
                   objFun = c(objFun, value),
                   iter = dim(convergeData)[3],
                   converged = converged,
                   convergeData = convergeData,
                   timing = t1)
    return(output)
}
```

II. Testings Performed

```
##############
# simulation
##############

#simulate toy dataset
library(mlbench)
n <- 100
```

```r
p <- 40
sigma <- 1
set.seed(30)
sim <- mlbench.friedman1(n, sd = sigma)
colnames(sim$x) <- c(paste("real", 1:5, sep = ""),
                     paste("bogus", 1:5, sep = ""))
bogus <- matrix(rnorm(n * p), nrow = n)
colnames(bogus) <- paste("bogus", 5 + (1:ncol(bogus)), sep = "")
x <- cbind(sim$x, bogus)
y <- sim$y

#binomial
#y[y < quantile(y, probs = 0.5)] <- 0
#y[y >= quantile(y, probs = 0.5)] <- 1

#poisson
sort(runif(10, min = 0, max = 10))/10
```

```
##  [1] 0.05371337 0.06136442 0.19363532 0.39706086 0.60920019 0.82394546
##  [7] 0.87751692 0.91589258 0.95834009 0.98257947
```

```r
dim(x);length(y)
```

```
## [1] 100  50

## [1] 100
```

```r
dim(x)
```

```
## [1] 100  50
```

```r
##############
#test function with simulated data
##############


a1 <- array(NA, dim = c(5, 4, 5))
a2 <- array(rnorm(100), dim = c(5, 4, 4))
a1[, , 2:5] <- a2

#GAfun(Y = iris$Sepal.Length, X = iris[ , - 1], objFun = "AIC")
#run against three different crossover/mutation methods
output.meth1 <- select(y, x, iter = 100, objFun = "AIC", parallel = F,
                crossMeth = "method1", converge = T, family = "gaussian")
```

```
## Loading required package: doParallel

## Warning: package 'doParallel' was built under R version 3.4.2

## Loading required package: foreach

## Loading required package: iterators

## Loading required package: parallel

## Loading required package: abind

## 1. Generate founders:  100 chromosomes
## 2. Evaluate founder fitness
## 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-
```

```
output.meth1.all <- select(y, x, iter = 100, objFun = "AIC", parallel = F,
                      crossMeth = "method1", converge = F) #with defaults
```

```
## 1. Generate founders:  100 chromosomes
## 2. Evaluate founder fitness
## 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-
```

```
output.meth2 <- select(y, x, iter = 100, objFun = "AIC", parallel = F,
                crossMeth = "method2", converge = T) #with defaults
```

```
## 1. Generate founders:  100 chromosomes
## 2. Evaluate founder fitness
## 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! ####
```

```
output.meth2.all <- select(y, x, iter = 100, objFun = "AIC", parallel = F,
                      crossMeth = "method2", converge = F) #with defaults
```

```
## 1. Generate founders:  100 chromosomes
## 2. Evaluate founder fitness
## 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-
```

```
output.meth3 <- select(y, x, iter = 100, objFun = "AIC", parallel = F,
                crossMeth = "method3", converge = T) #with defaults
```

```
## 1. Generate founders:  100 chromosomes
## 2. Evaluate founder fitness
## 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! ####
```

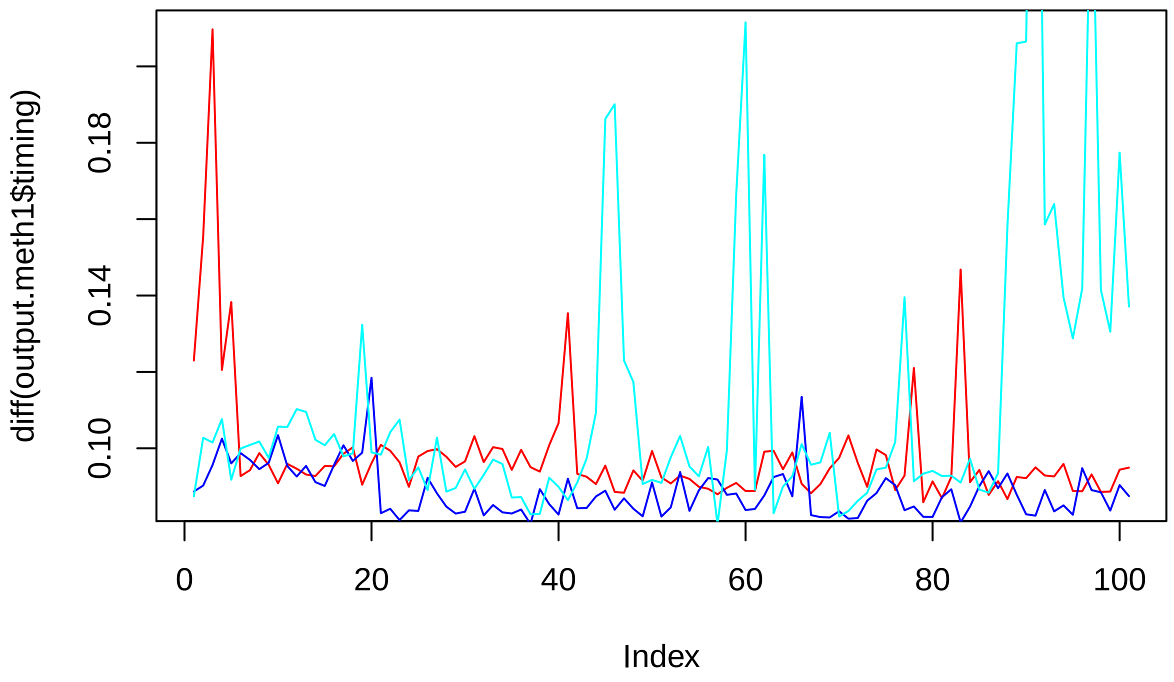
```
output.meth3.all <- select(y, x, iter = 100, objFun = "AIC", parallel = F,
                      crossMeth = "method3", converge = F) #with defaults
```

```
## 1. Generate founders:  100 chromosomes
## 2. Evaluate founder fitness
## 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-
```

```
test <- rbind(c(output.meth1$objFun, output.meth1$iter),
        c(output.meth1.all$objFun, output.meth1.all$iter),
        c(output.meth2$objFun,     output.meth2$iter),
        c(output.meth2.all$objFun, output.meth2.all$iter),
        c(output.meth3$objFun,     output.meth3$iter),
        c(output.meth3$objFun,     output.meth3.all$iter))

colnames(test) <- c("ObjFun", "value", "iterations")
rownames(test) <- grep("output.meth", ls(), value = T)
test
```

```
##                  ObjFun value              iterations
## output.meth1     "AIC"  "160.651597184073" "101"
## output.meth1.all "AIC"  "158.978441313556" "101"
## output.meth2     "AIC"  "156.132943113861" "44"
## output.meth2.all "AIC"  "152.172002019561" "101"
## output.meth3     "AIC"  "154.293899127272" "60"
## output.meth3.all "AIC"  "154.293899127272" "101"
```

```
#timing
plot(diff(output.meth1$timing), type = "l", pch = 20, col = "red")
lines(diff(output.meth2.all$timing), pch = 20, col = "blue")
lines(diff(output.meth3.all$timing), pch = 20, col = "cyan")
```



```
mean(diff(output.meth1.all$timing))
```

```
## Time difference of 0.09434945 secs
```

```
mean(diff(output.meth2.all$timing))
```

```
## Time difference of 0.08860644 secs
```

```
mean(diff(output.meth3.all$timing))
```

```
## Time difference of 0.1124424 secs
```

```
sum(diff(output.meth1.all$timing))
```

```
## Time difference of 9.529294 secs
```

```
sum(diff(output.meth2.all$timing))
```

```
## Time difference of 8.949251 secs
```

```
sum(diff(output.meth3.all$timing))
```

```
## Time difference of 11.35669 secs
```

```
#check variables selected
colnames(x)[output.meth1$BestModel == 1]
```

```
##  [1] "real1"   "real2"   "real4"   "real5"   "bogus7"  "bogus8"  "bogus9"
##  [8] "bogus12" "bogus16" "bogus18" "bogus19" "bogus24" "bogus25" "bogus27"
## [15] "bogus28" "bogus30" "bogus31" "bogus32" "bogus33" "bogus35" "bogus40"
## [22] "bogus41" "bogus44" "bogus45"
```

```
colnames(x)[output.meth2$BestModel == 1]
```

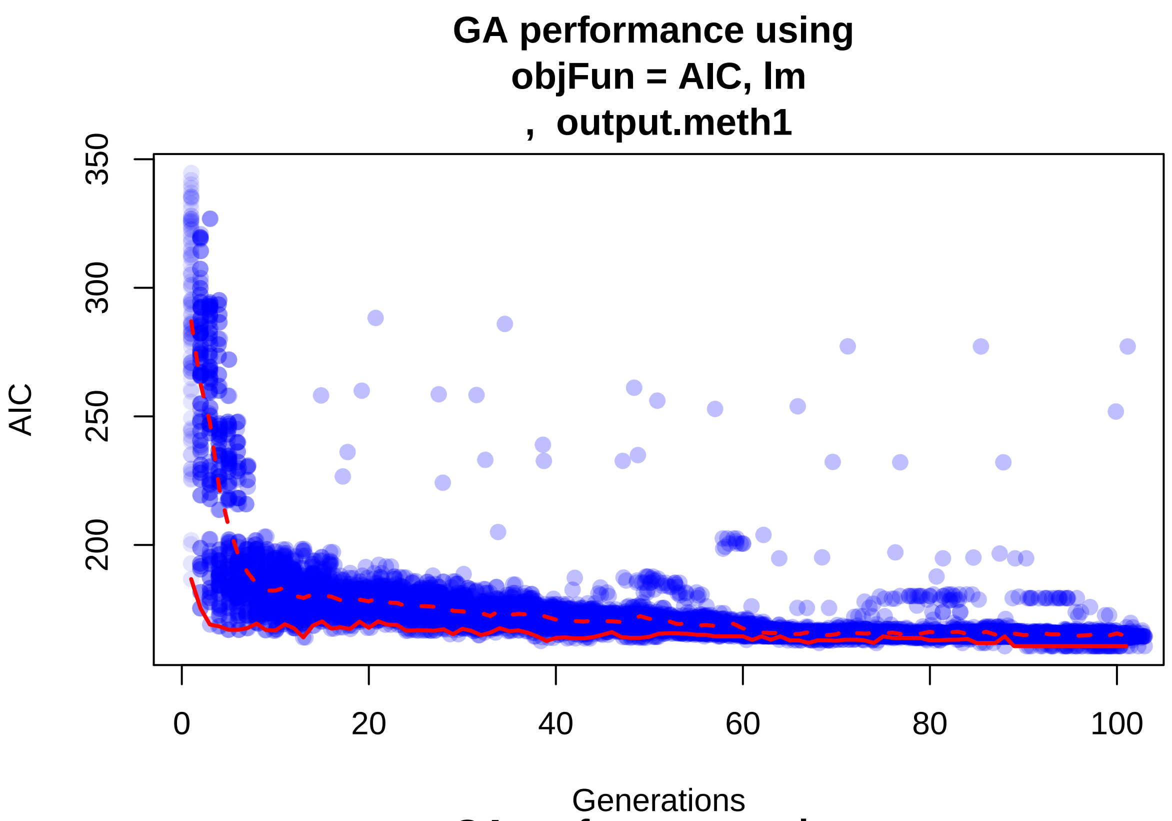
```
##  [1] "real1"   "real2"   "real4"   "real5"   "bogus5"  "bogus7"  "bogus8"
##  [8] "bogus9"  "bogus16" "bogus20" "bogus24" "bogus28" "bogus30" "bogus31"
## [15] "bogus35" "bogus36" "bogus40" "bogus44" "bogus45"
```

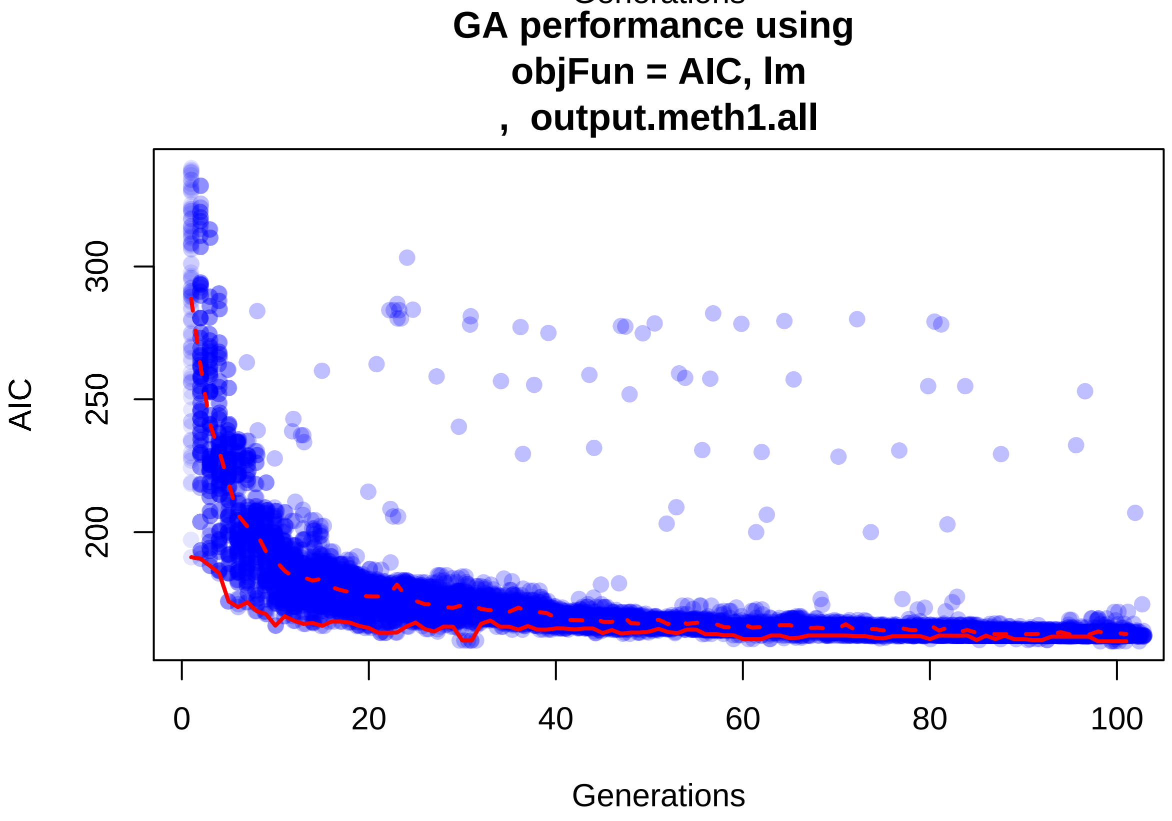
```
colnames(x)[output.meth3$BestModel == 1]
```

```
##  [1] "real1"   "real2"   "real4"   "real5"   "bogus7"  "bogus8"  "bogus9"
##  [8] "bogus11" "bogus16" "bogus20" "bogus24" "bogus25" "bogus28" "bogus30"
## [15] "bogus35" "bogus40" "bogus44" "bogus45"
```

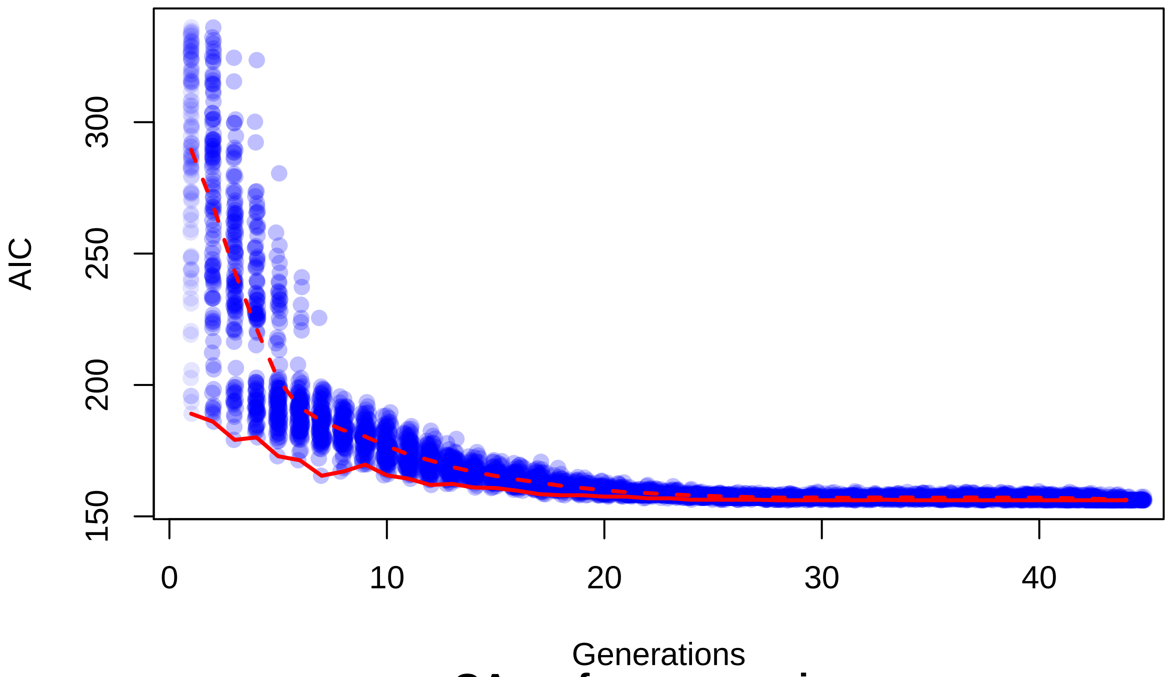
```
#plot results
plots <- grep("output.meth", ls(), value = T)
for (i in 1:length(plots)) {

    output <- eval(parse(text = plots[i]))
    convergeData <- output$convergeData
    require(scales)

    #png(paste0("~/repos/STAT243/project/GAplots_method_0", i, ".png"),
    #    width = 640, heigh = 480)

    par(mfrow = (c(1, 1)))

        #all AIC across interations
        plot(jitter(rep(1, nrow(convergeData))), convergeData[, 2, 1], type = "p",
               pch = 19, col = alpha("blue", 0.1),
               ylim = c(min(convergeData[, 2, ], na.rm = T),
                        max(convergeData[, 2, ], na.rm = T)),
               xlim = c(1, output$iter), xlab = "Generations", ylab = "AIC",
               main = paste("GA performance using \n objFun = AIC, lm \n, ",
                            plots[i]))
        for (i in 2:output$iter) {
            points(jitter(rep(i, nrow(convergeData))), convergeData[, 2, i], type = "p",
                   pch = 19, col = alpha("blue", 0.25))
        }

        #best AIC per iteration
        lines(1:output$iter, sapply(1:output$iter,
                                    function(x) convergeData[1, 2, x]), type = "l",
             xlab = "iterations", ylab = "AIC", col = "red", lwd = 2,
             main = paste("Best AIC per iteration \n objFun = AIC, lm \n",
                          plots[i]))
        lines(1:output$iter, sapply(1:output$iter,
                                    function(x) mean(convergeData[, 2, x])),
                                    lty = 2, lwd = 2, col = "red")

    #dev.off()
}
```

```
## Loading required package: scales
```

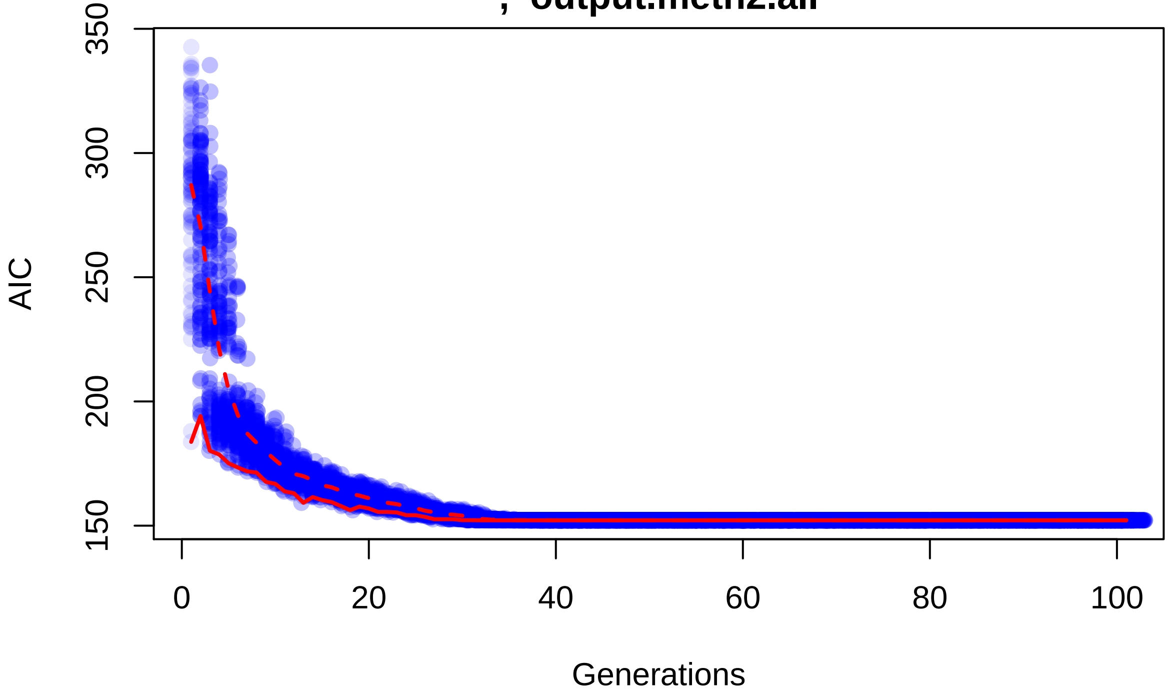

GA performance using
objFun = AIC, lm
, output.meth1
AIC
350
300
250
200
0
20
40
60
80
100
Generations

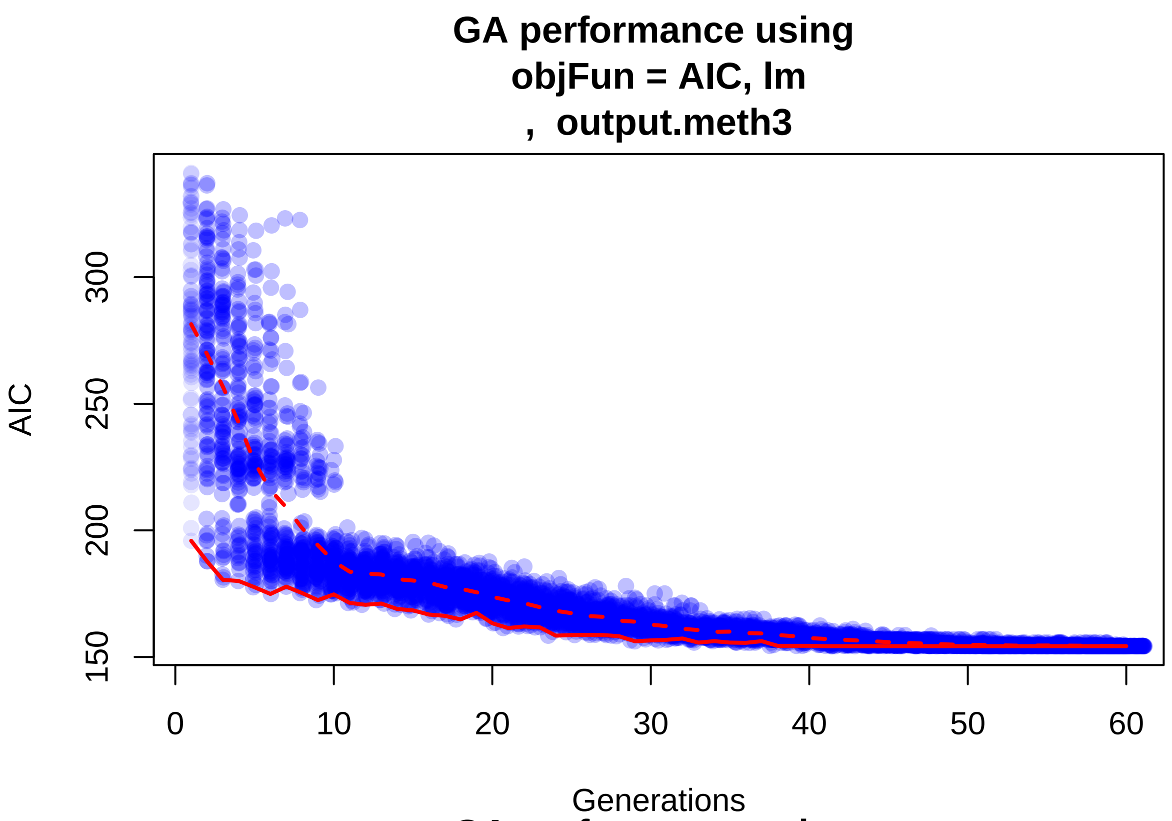

GA performance using
objFun = AIC, lm
, output.meth1.all
AIC
300
250
200
0
20
40
60
80
100
Generations

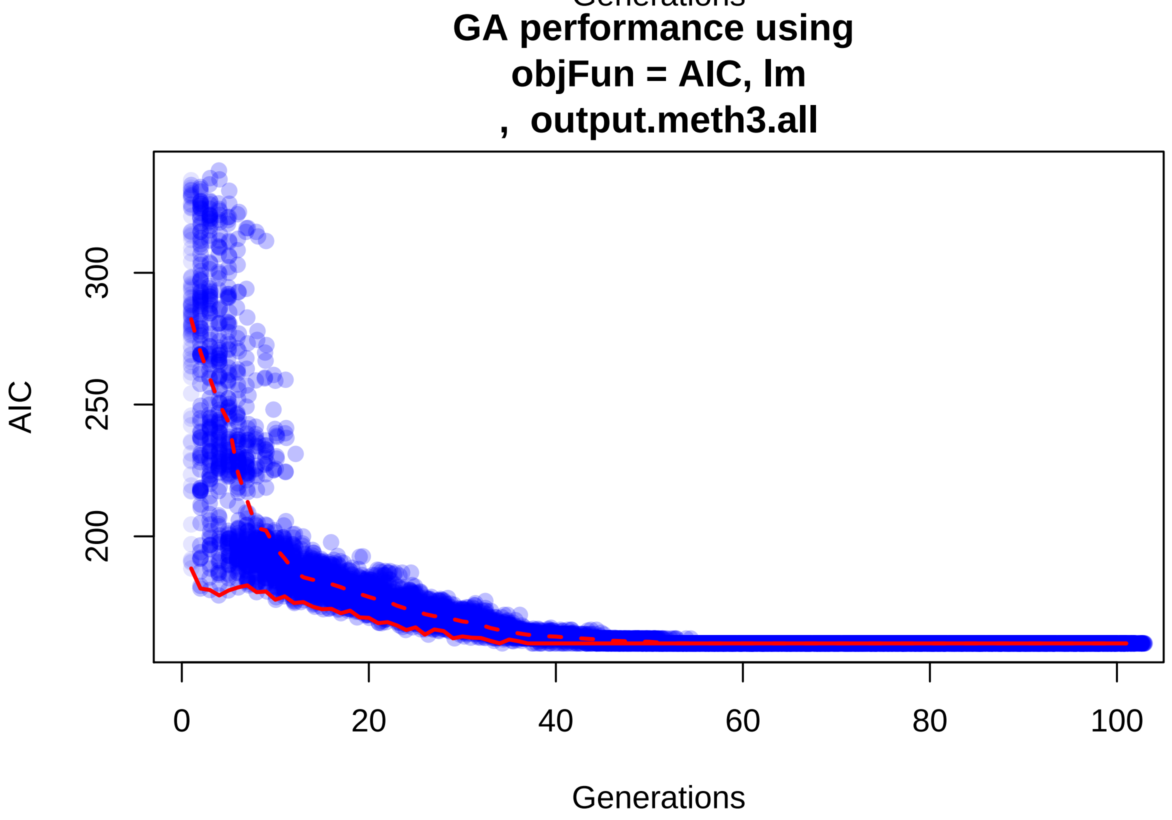
GA performance using objFun = AIC, lm , output.meth2



GA performance using objFun = AIC, lm , output.meth2.all




GA performance using
objFun = AIC, lm
, output.meth3
AIC
150
200
250
300
0
10
20
30
40
50
60
Generations


GA performance using
objFun = AIC, lm
, output.meth3.all
AIC
200
250
300
0
20
40
60
80
100
Generations

Real Implementations

Contributions of Members

Reference