STAT 243 Final Project Report

Eric Kim, Shuyu Zhao, Rui Chen, Hangyu Huang
Github Repository: erickim/GA

December 14, 2017

0 Installation

In order to install the package and load it into R, please run the following. Ignore the first line if you already have `devtools`.

```r
library(devtools)
devtools::install_github("erickim/GA")
library(GA)
```

1 Functionality and Procedure

The Genetic Algorithms (GAs) are stochastic search algorithms that mimic the process of Darwinian natural selection. GAs simulate the biological evolution, where breeding among highly fit organisms ensures desirable attributes be passed to future generations, thereby providing a set of increasingly good candidate solutions to the optimization.

The select function enables the application of genetic algorithms to problems where the decision variables are encoded as "binary".

Selection mechanism mimics the process by which parents are chosen to produce offspring. Crossover and mutation operations are used to produce offspring chromosomes from chosen parent chromosomes.

Rank-based method is applied here to prevent GAs convergence to a poor local optimum, and parents are chosen based on the rank of values of negative AIC function. Any R function, which takes as input an individual string representing a potential solution, that returns a numerical value describing its "fitness" is allowable to perform as a fitness function.

The population size is in the range of the chromosome length to two times of chromosome length, though this can be overridden by the user. In this function, the default for the population size is twice of chromosome length, which is the number of columns of the feature matrix.

Our solution consists of one main `select` function which will iterate over a user specified number of generations and return the most fit individual. It will make calls to helper functions, which we describe below.

1.1 Utility Functions

The following functions can be found in `./R/utils.R`.

1.1.1 `initialize`

The function `initialize` is used to create P initial parents to start the genetic algorithm. In this function, we pass in the response vector and the feature matrix along with the number of candidates in the population as well as the type of regression. It will create 0-1 vectors to indicate which features are included in each candidate for the initial population, and return these vectors along with the corresponding regression fit.

1.1.2 `crossover`

The `crossover` function gives a means to create two new children from two parents by swapping the segments of parents' chromosomes. We can have one point to split at, which means the chromosomes are divided into two segments and then combined with each other. We can also have multiple points to split at, and the chromosomes will combine in an alternating fashion. It will return a list with two vectors for each child.

1.1.3 `mutate`

The `mutate` function is used to get the child gene after mutation. In this function, we take in a mutation rate along with the child to mutate and create a 0-1 vector representing the child with each entry potentially mutated.

1.1.4 `selection`

`selection` is used to select parents from current generation that will produce offsprings. In this function, we provide two different methods of selection based on the probability of being chosen as a parent. The probabilities are calculated as the fitness of the individuals devided by the sum of fitness value of the whole population. The "oneprop" type randomly selects one parent based on the fitness probability and the second parent randomly with equal chance from the remaining population. The "twoprop" type will randomly select both parents based on the fitness probabilities. Note that tournament selection was one of our to-do's but we never got around to it.

1.1.5 `fitnessRanks`

The `fitnessRanks` function will take in a vector of fitnesses and return the ranked version. This is primarily used when we ignore original fitness values and instead use their ranks.

1.1.6 `regFunc`

The `regFunc` function is a nice wrapper for `lm` or `glm`. Based on the regression type and family if the type was `glm`, `regFunc` will take in a formula and the dataset and produce the appropriate fitted model.

1.2 `select`

The main `select` function will first initialize the first generation by calling `initialize` function. If the fitness function was left at the default of AIC, then we create a new fitness function that evaluates the negative AIC (so that higher fitness means better candidate). Otherwise we will just use the user supplied fitness function leaving it up to the user to correctly implement such a function.

We will further allow the user to specify `elitism`. If it is requested, at the start of each generation, the best candidate from the previous generation will automatically be inherited to the current generation and we will keep track of how long the most elite candidate has been carried over. We do not specify a particular convergence rule and instead allow the user to supply the maximum number of iterations as suggested by Givens/Hoeting. So the length of elitism can be used as a measure of convergence: if it is high then it means that the elite individual seems to be the best as none of the generated children are able to beat it.

Next, we start a loop to create each generation. First the fitness functions will be collected. Then the `selection` function will be called to get the new parents. `crossover` and `mutate` will then be called to generate the new children. These children along with their fitted regression model will be added to the population. We add a check where if after the first child is added our population is full at P, then we disregard the second child and move on.

If `elitism` was requested, we will compare the elite candidate from the previous generation to the current generation. If they were the same, then the length of elitism will be updated by 1. This will continue until the maximum number of iterations is reached at which point the variables, fitted model, fitness, fitness type, and length of elitism will be returned.

This can be summarized with the diagram in Figure 1.

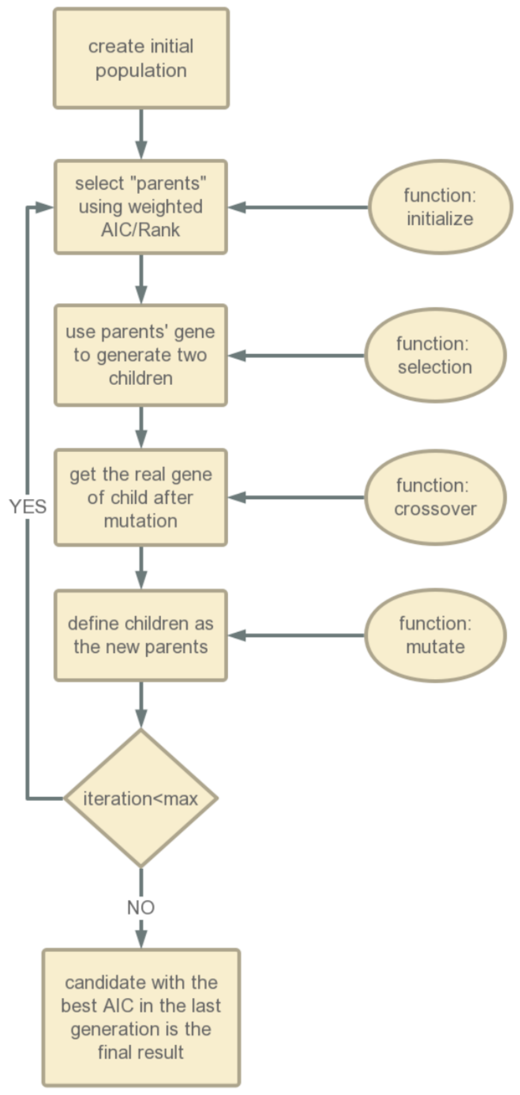
create initial population
select "parents" using weighted AIC/Rank
function: initialize
use parents' gene to generate two children
function: selection
get the real gene of child after mutation
function: crossover
YES
define children as the new parents
function: mutate
iteration<max
NO
candidate with the best AIC in the last generation is the final result

Figure 1: Flow chart of `select`.

2 Testing

We will test each of the modular pieces as well as the overall `select` function. We will make fake data as necessary to test implementation and either the `mtcars` or `Boston` datasets as we see fit for regressions. All of the following can be run using `test_package("GA")`. There will be a couple warnings, but the tests will all pass.

2.1 `initialize`

Here, we look at the `mtcars` dataset. We test that when a bad argument passed in for Y or P or seed, the function will error out while when a bad argument is passed in for regType, it will default to lm and continue. We also test that when correct arguments are passed in we will get P items.

```
Y <- mtcars$mpg
X <- mtcars[2:11]
P = 2 * ncol(X)
regType = 'lm'
family = 'gaussian'
seed = 1

test_that("input is invalid",{
  # Y is not valid
  expect_error(initialize(Y = "a", X = X, P = P, regType = regType,
                          family = family, seed = seed))
  # P needs to be a numeric
  expect_error(initialize(Y = Y, X = X, P = "a", regType = regType,
                          family = family, seed = seed))
  # if bad regType, it will auto default to 'lm'
  expect_is(initialize(Y = Y, X = X, P = P, regType = 1,
                       family = family, seed = seed), "list")
  expect_is(initialize(Y = Y, X = X, P = P, regType = P,
                       family = 1, seed = seed), "list")
  expect_error(initialize(Y = Y, X = X, P = P, regType = regType,
                          family = family, seed = "a"))
  expect_equal(length(initialize(Y, X, P, regType, family, seed)),
               2 * ncol(X) )
})
```

2.2 `crossover`

Here we test when bad inputs are put into `crossover`. It will either throw an error or print a message and return NA. We also create some fake data just to test that the crossover operation actually works.

```
test_that("input is not valid", {
  # incorrect inputs are caught
  expect_equal(crossover(c(2,0),1), NA)
  # invalid crossover type
  expect_error(crossover(1, 1, type = "abc"))
})
# create some fake data to test crossover
parent1 <- rbinom(20, 1, 0.5)
```

```
parent2 <- rbinom(20, 1, 0.5)
result1 <- crossover(parent1, parent2, type = "single", num_splits = 1)
result2 <- crossover(parent1, parent2, type = "multiple", num_splits = 5)
test_that("ouput is not expected", {
  # makes sure we get correct outputs for single splits
  expect_equal(length(result1$child1), 20)
  expect_equal(length(result1$child2), 20)
  expect_false(any(result1$child1 != 0 & result1$child1 != 1))
  expect_false(any(result1$child2 != 0 & result1$child2 != 1))
  # makes sure we get correct outputs for multiple splits
  expect_equal(length(result2$child1), 20)
  expect_equal(length(result2$child2), 20)
  expect_false(any(result2$child1 != 0 & result2$child1 != 1))
  expect_false(any(result2$child2 != 0 & result2$child2 != 1))
})
```

2.3 mutate

Here, we test that bad inputs for `mutate` will error out. For correct inputs, we just check that the output is of expected length.

```
rate<-0.2
offspring<-c(1,1,1,1,0,0,0,0)

mutate(rate, offspring)
## [1] 1 1 1 0 0 1 0 0
test_that("input is invalid",{
  expect_error(mutate("a", offspring))
  expect_error(mutate(2, offspring))
  expect_equal(length(mutate(rate, offspring)), length(offspring))
})
```

2.4 selection

Here we test that when bad fitnesses are passes as arguments, we will just get an NA. We also test that when we pass in fine values for the type it will work as expected while if we pass in bad arguments, it will default to "twoprop" and continue.

```
test_that("Test if the input of selection is valid", {
  expect_equal(selection("oneprop",  c("a","a","a","a","a")), NA)
  expect_equal(selection("oneprop",  2), NA)
  expect_equal(selection("twoprop", c(2,3, "w" , "w")), NA)
  expect_equal(selection("twoprop", 2), NA)

})

test_that("Test if the output of selection is valid", {
  expect_equal(length(selection("oneprop", c(0.9,1.5,3.3,2.2))), 2)
  expect_equal(length(selection("twoprop", c(0.9, 1.5, 3.3, 2.2))), 2)
  expect_equal(length(selection("hehehe", c(0.9, 1.5, 3.3, 2.2))), 2)
  expect_true(is.numeric(selection("oneprop", c(0.9, 1.5, 3.3, 2.2))))
  expect_true(is.numeric(selection("twoprop", c(0.9,0.6,0.1,1.1))))
})
```

2.5 fitnessRanks

Here, we just check that the `fitnessRanks` function is ranking properly and that it will error out if a bad argument is passed.

```
fitness <- 1:10
fitnessRanks(fitness)

##  [1] 0.01818182 0.03636364 0.05454545 0.07272727 0.09090909 0.10909091
##  [7] 0.12727273 0.14545455 0.16363636 0.18181818

test_that("input is invalid",{
  expect_equal(fitnessRanks("a"), NA)
  expect_equal(length(fitnessRanks(fitness)),length(fitness))
})
```

2.6 regType

Here we test that the `regType` will perform the proper regression and we use the Boston crimes dataset from the `MASS` package to fit the model.

```
library(MASS)
# use the Boston crime dataset to check that the regression function wrapper
# works as it should
boston.crim = Boston$crim
test_that("Test if the output of regFunc is valid",{
  expect_equal(regFunc("lm",
                       "gaussian",
                       boston.crim ~.,
                       Boston[,-1])$Coefficients,
               lm(crim~., data = Boston)$Coefficients)
  expect_equal(regFunc("glm",
                       "gaussian",
                       boston.crim ~., Boston[,-1])$Coefficients,
               glm(crim~., data = Boston, family = "gaussian")$Coefficients)
})
```

2.7 select with Boston

To test whether our implementation of `select` is correct, we use the `Boston` dataset again. From the `leaps` package, we use the `regsubsets` function to perform a best subset search for all models with 1 through 13 features (13 being the maximum number of said features). We find the best AIC for these 13 best models and compare it to the final output of the `select`. We just test that both AIC's are relatively close to each other (they turn out to be the same as they picked the same models).

```
# do a best subsets selection first
bostSubsets <- regsubsets(crim ~ ., data = Boston, nvmax = 13)
bostSubsetsWhich <- summary(bostSubsets)$which
```

```r
bostSubsetsAIC <- c()

for (i in 1:13) {
  bostSubsetsAIC <- c(bostSubsetsAIC,
                      AIC(glm(crim ~ .,
                              data = Boston[c(TRUE,
                                              bostSubsetsWhich[i,-1])])))
}

# maxIter = 50 because the dataset isn't too big
bostGA <- select(Boston$crim, Boston[,-1], regType = "lm", maxIter = 50)
min(bostSubsetsAIC)
## [1] 3329.8
test_that("`select` comes close to the best subset selection", {
  expect_equal(-1*bostGA$fitness, min(bostSubsetsAIC),
               tolerance = min(bostSubsetsAIC)/1e5)
})

# compare the selected variables
which(bostGA$variables == 1)
## [1]  1  4  7  8 10 11 12 13
which(bostSubsetsWhich[which.min(bostSubsetsAIC),-1])
##      zn     nox     dis     rad ptratio   black   lstat    medv
##       1       4       7       8      10      11      12      13
```

2.8 select with swiss

We do the same procedure as above except with the Swiss Fertility and Socioeconomic Indicators dataset, **swiss**, in the "datasets" package and has 47 observations on 6 variables. We regressed Fertility on other aspects to find the influencing factors of Swiss Fertility and try different values of maxIter. It seems that we needed more than 5 iterations but no more than 10 so for these small datasets, convergence is relatively quick as expected.

```r
#Swiss Fertility and Socioeconomic Indicators (1888) Data
Y <- swiss$Fertility
X <- swiss[,-1]

#use select function to select variables
GASelect5 <- select(Y,X, maxIter = 5)
GASelect10 <- select(Y,X, maxIter = 10)
GASelect15 <- select(Y,X, maxIter = 15)
GASelect25 <- select(Y,X, maxIter = 25)
GASelect50 <- select(Y,X, maxIter = 50)

#use regsubsets to select variables
testBest <- regsubsets(Y ~ ., data = X, nvmax = 5)
testBestWhich <- summary(testBest)$which
testBestAIC <- c()
for (i in 1:5) {
  testBestAIC <- c(testBestAIC,
                   AIC(glm(Y ~ .,
                          data = X[testBestWhich[i,-1]])))
}

# compare the selected variables (not using testthat)
which(GASelect5$variables == 1)
## [1] 1 2 3 4 5
which(GASelect10$variables == 1)
## [1] 1 3 4 5
which(GASelect15$variables == 1)
## [1] 1 3 4 5
which(GASelect25$variables == 1)
## [1] 1 3 4 5
which(GASelect50$variables == 1)
## [1] 1 3 4 5
which(testBestWhich[which.min(testBestAIC),-1])
##      Agriculture        Education         Catholic Infant.Mortality
##                1                3                4                5
```

3 Contributions

Eric: designed the structure of the package, wrote some of the helper functions, wrote `select`, and helped write the documentation.

Rui: wrote some of the helper functions, debugged the functions, wrote the test and example code, and contributed to the final writeup.

Shuyu: wrote some of the helper functions, debugged the functions, wrote the test and example code, and contributed to the final writeup.

Hangyu: wrote some of the helper functions, debugged the functions, wrote the test and example code, and helped write the documentation.