STAT 243 Final Project: Genetic Algorithm Documentation

Kunal Desai, Fan Dong, James Duncan, Xin Shi

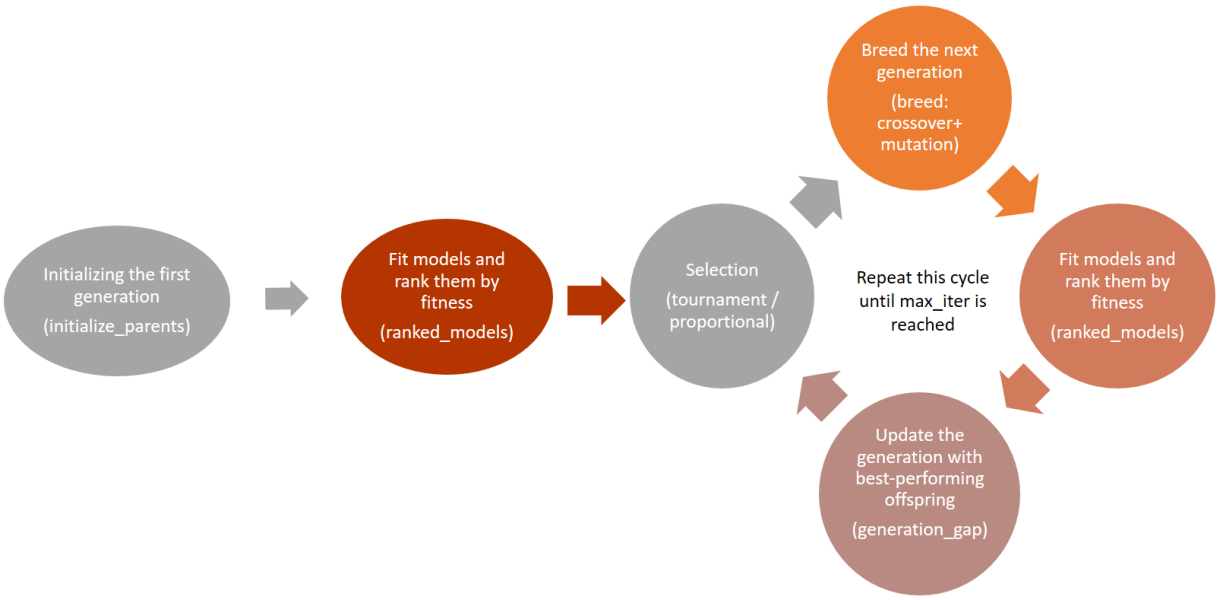
December 14, 2017

Installing and Testing

```
library(devtools)
install_github("kunaljaydesai/GA")
library(GA)
library(testthat)
test_package("GA")
```

How *select* Works

Modularity and Approach



The Genetic Algorithm package is designed modularly, each step with auxiliary functions accomplishing discrete tasks.

As the flowchart above shows, the algorithm consists of six steps. First, through *initialize_parents*, we setup the first generation of P models by randomly selecting features for each member of the generation. Once that was completed, we calculate the fitness of each model inside the generation and rank all the models by their fitness (using *calculate_fitness* and *ranked_models*).

Next, we enter the loop and start the first selection process of picking the pairs of parents for the next generation. There are three selection mechanism: proportional, rank or tournament. The first two methods use *proportional*. When calling proportional selection (set random=TRUE), one parent is picked proportional to its fitness and the other picked completely randomly; when calling rank selection (set random=FALSE), both parents are picked proportional to their fitness. Otherwise selection was chosen to be tournament selection (use *tournament*) in which every time we randomly select k members from the generation and the best in each round becomes a parent. Once the parents have been chosen, the children need to be created (use *breed*) via cross over. Then, to increase diversity, there is a 1% possibility that the expression of a feature will be randomly altered. Once the children are selected, like their parents, they are ranked by fitness (*ranked_models*). Next, using *generation_gap*, we replace the n worst individuals with n new individuals from the old generation. This is the final step in determining the new generation. Once this is complete, we identify the member in the generation with the best fitness and, if better than any individual seen thus far, we make that the overall best member.

We repeat this process until the convergence criteria is met. In our case, the criteria is the maximum number of iterations, which can either be specified by the user or set at a default of 100.

To summarize, when calling the primary function *select*, the following is happening under the hood:

select: put all the functions together while iterating till reaching convergence criteria

- *initialize_parents*: set up the initial generation
- *ranked_models*: rank all the models based on their fitness value
  - *calculate_fitness*: calculate the fitness (AIC by default) of a given feature set
- *tournament*, *proportional* (random = TRUE / FALSE): select parents out of the current generation
- *breed*: take a generation and output its children, mutating some features with a low probability (1% by default)
  - *crossover*: take in a list of places to split (number of splits can be specified by the user or 2 by default) and create a set of children
- *generation_gap*: determine which parents will belong in the new generation and which members of the new generation will be kicked out

Testing

In terms of testing, we first ensured that all functions were tested for proper inputs. This includes auxiliary functions that aren't intended for public use. We did input sanitization for all functions to ensure that it would gracefully handle mal-formed error including non-integer/float inputs and NA inputs. We also had to write tests for inputs that didn't make sense in relation to the function. For example, if the number of crossover points chosen was greater than the length of the chromosome. Finally, we also ensured that all of our tests worked for inputs we expected it to work on. We also checked for the accuracy of our results in our test cases. When writing our code, we used a pair programming approach to limit defects in the code. We also would write a function and have someone else write tests for it to ensure we could catch as many cases as possible.

An Example

To test the algorithm, we randomly generate a dataset of 10 predictors and 20 observations, in which 4 are real predictors (named $real_1 \sim real_4$) and the rest are pure noise variables($noi_1 \sim noi_6$). The response variable, y, is thus given by the equation (coefficients are picked randomly):

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$

Code for data simulation:

```
library(GA)
set.seed(1)
x <- matrix(rnorm(10*20),ncol=10) # generate a data with 10 features
beta <- c(1,3,5,7)
y <- as.vector(x[,1:4] %*% beta) # the true feature is first four features
colnames(x) <- c(paste("real", 1:4, sep = ""),
                 paste("noi", 1:6, sep = ""))
select(x,y) # By GA algorithm
```

```
## $survivor
## [1] 1 2 3 4
##
## $fitness
## [1] -1267.854
##
## $num_iteration
## [1] 100
##
## $first_seen
## [1] 5
```

```
permutations <- initialize_parents(10,1024) # genearte all possible models
result <- ranked_models(permutations$index,x,y)
result[which(result$fitness == min(result$fitness)),] # find the best model with highest fitness
```

```
##        Index   fitness
## 1 1, 2, 3, 4 -1267.854
```

Our genetic algorithm was able to successfully find the global optimum in this small example.

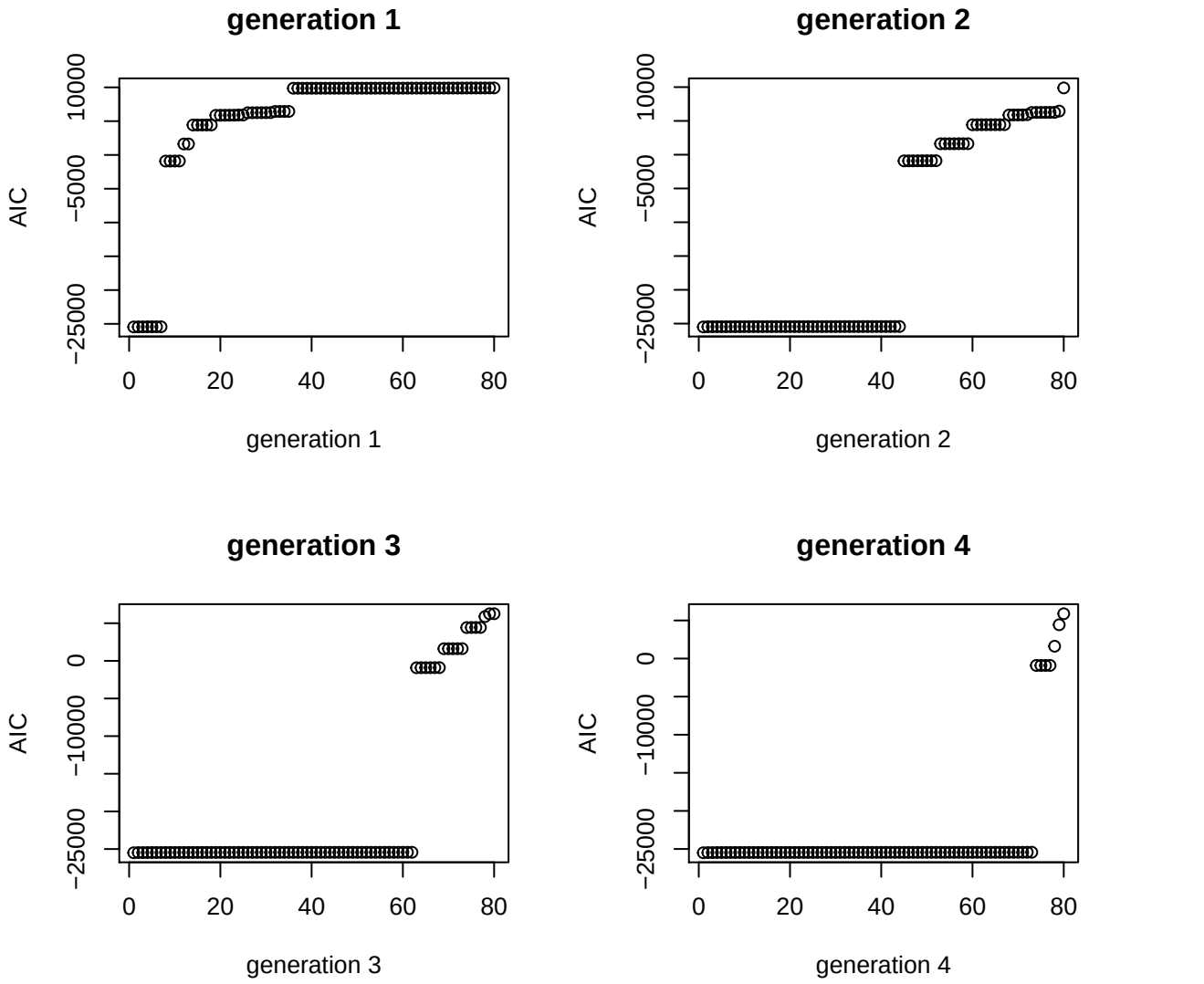
Here is a larger example with 500 observations and 40 features, 6 of which are the true covariates. We will look at the fitness evolution from generation to generation.

```
par(mfrow = c(2,2))
##### generate data with 40 features, 500 observations ######
n <- 500
c <- 40
X <- matrix(rnorm(n * c), nrow = n)
beta <- c(88, 0.1, 123, 4563, 1.23, 20)
y <- as.vector(X[ ,1:6] %*% beta)  # true model is first 6 features
colnames(X) <- c(paste("real", 1:6, sep = ""),
                 paste("noi", 1:34, sep = ""))
select(X,y)
```

```
## $survivor
## [1]  1  2  3  4  5  6 26
##
## $fitness
## [1] -25510.31
##
## $num_iteration
## [1] 100
```

```
##
## $first_seen
## [1] 72
```

```
initial <- initialize_parents(40,80) # initialize to generate index
old_gen <- ranked_models(initial$index,X,y) # lm each model
for( i in 1:4){
  ### choose parents from old gen
  parents <- propotional(old_gen, random = T)
  ### crossover and mutation
  children <- unique(unlist(lapply(parents, breed,C = 40),FALSE, FALSE))
  ### lm children
  ranked_new <- ranked_models(children, X, y)
  ### generation gap
  next_gen <- generation_gap(old_gen, ranked_new)
  plot(old_gen$fitness,main = paste("generation",i),ylab = "AIC", xlab =paste("generation",i))
  old_gen <- next_gen
}
```



From the graph above, we can see that in each iteration, generation is improved and more fitted models takes a larger proportion of generation.

How the Team Works

The project resides in Kunal's repository (Git Username: kunaljaydesai, Repo name: GA).

The specific tasks completed by each group member is listed below:

- Kunal

Wrote the initialize parents function and calculate fitness function. Wrote tests for respective functions and created testing pipeline (directory and autotester). Created framework for package including roxygen2 documentation setup. Wrote *Modularity and Approach* and *Testing* section in this documentation.

- Fan

Wrote ranked_models, tournament and their respective tests. Modified and finalized calculated fitness. Finalized roxygen2 documentation for all functions. Modified and finalized *Modularity and Approach* section in this documentation. Wrote the *An Example* section in this documentation with Xin.

- James

Wrote breed and crossover. Wrote generation_gap with Fan, and worked on select. Wrote tests for breed, crossover, generation_gap, and select. Proofread and finalized this documentation.

- Xin

Wrote propotional function and select function. Test and improve the main function to converge better and faster with James. Wrote *Example* section in this documentation with Fan.