STAT 243 : Project

Skander Jemaa
Rivers Jenkins
Yachuan Liu
Yu Wang

December 14, 2017

GitHub

The package is located in Skander Jemaa's (skjemaa) github at https://github.com/Skjemaa/GA.

Introduction

We were asked to implement a genetic algorithm for regression for this package. We chose a functional coding style. The main function the user will interface with is the select() function which calls many utility functions also contained in the package.

We identified 3 main parts in the implementation of the genetic algorithm. The first one was the parents selection mechanism, then came the gene operator and the last one was to carry out the production of a new generation.

For each part, we have designed several approaches and the arguments can be changed by the user that can provide their own functions.In order to ease up reading and debugging we designed utility functions for each step.

We will present each step of the genetic algorithm below followed by a description of the main select function which initiates the genetic algorithm.

Parent Selection Mechanism

To create each successive generation in the genetic algorithm, we have to select parents from the previous generation to be paired off. After two parents are paired off, they will create two children according the the gene selection mechanisms detailed in a later section. In order to select parents for each generation, there are a variety of methods available to us. We chose to implement the following four methods of parent selection.

1. Random: This selection mechanism is the simplest. We randomly choose two parents from the population to pair off.

2. Proportional: This selection mechanism is more discriminating in picking parents. For each individual in the current generation, the fitness or objective function is calculated. We then divide this value by the sum of the objective values for all the individuals in that generation to get the probability of selecting an individual to be a parent. This allows the the algorithm to pick two parents proportionally to their fitness values. Both parents are picked in this manner.

3. Proportional/Random: This method is a hybrid of the previous two methods. One parent is selected according to the proportional method described above, and one parent is selected uniformly randomly.

4. Tournament: This selection mechanism groups the individuals of the current generation in groups of size k, and then selects parents based on the best fitness within each of these groups.

Gene operator

After the parents are selected for a new generation, we must create the new children for this generation. We can do this using a variety of gene operators on the two parents. The gene operators we implemented are detailed below.

1. Crossover: This is similar to selecting which chromosome to take from each parent in human reproduction. We select a sequence of alleles (in this case binary digits) from each parent to add to the child. So if each parent has 5 alleles with one as (1,0,0,0,0) and the other as (1,1,1,1,1), we could take the first 3 alleles from parent 1, and the last two from parent 2 giving (1,0,0,1,1). These sequences do not have to be contiguous, thought they were in that example.

2. Allele Swapping: This is similar to crossover, but instead of looking at sequences of alleles, we swap individual alleles. After the initial children are formed using crossover, we randomly pick k alleles to swap between the children. So (1,1,0,0) could become (1,1,0,1) if we pick the last allele to swap and the other child had a 1 in that position.

3. Mutation: This is the final step in constructing the new children and is performed separately on each child. We will randomly mutate an allele according to some (usually quite small) probability. This ensures that we are not totally restricted by the alleles in the parent generation.

Iterations of the genetic algorithm

Initialization

In order to initialize the algorithm, we have to create the first parent generation from scratch. Since we knew that the genetic algorithm would only be applied to regressions, we also wanted to expand the tools of the user by adding an option to look at interactions between covariates. This greatly expands the pool of variables.

In order to accomplish the initialization of the first generation we radomly select which variables to include in the first regressions.

Randomly selecting variables for the first generation gives good diversity in the first generation, but it may also mean we are far from the optimal solution and will take a long time to converge. In order to avoid this, we provide an option for the user to only consider the most significant variables in the dataset. By most significant, we mean that we only consider those variables with a p-value of less than 0.05 when run in a simple linear regression. This way we restrict the variable population at the beginning in order to get faster convergence.

Iterations

In each iteration of the algorithm we select new parents from the current generation and use those to create the next generation, stopping if the algorithm seems to be converging (we see a little change in the objective value). However, we may wish to keep the best performing individuals in the current generation. We can accomplish this by imposing a permutation gap. The permutation gap will decrease the overall population each iteration by chopping off the most unfit individuals. This new population will survive to the next generation, while the worst performers are replaced by children. The permuation gap is defined in our algorithm as the proportion of individuals to kill off in each each iteration.

After the population is whittled down to size, the parents are selected, and a new generation is created.

Select function

Our main function in our package is select() which has many parameters that can be modified by the user to obtain optimal performance from the algorithm. The three required parameters are the dependent variable y, the dataset of covariates with the last column corresponding to y, and the regression model (either "lm" or "glm"). These are required for any call to select.

The optional parameters are detailed below.

1. n_iter: the maximum number of iterations for the algorithm to run

2. pop_size: the initial population size

3. objective: the objective criteria to use (default is "AIC")

4. interaction (logical): Whether to add the interaction terms to the independent variables.

5. most_sig (logical): Whether to use the most significant variables inside the first_generation function.

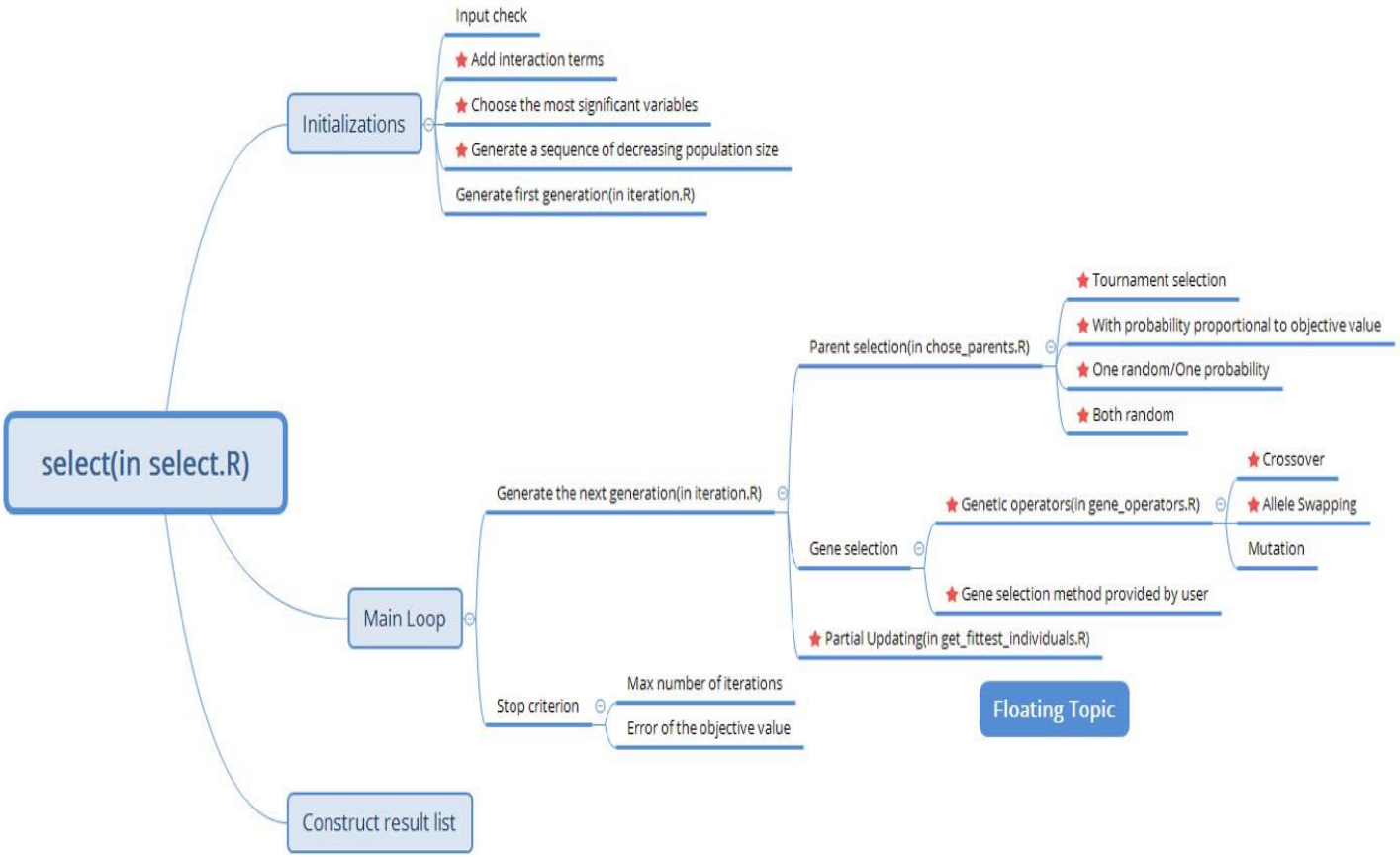
6. parent_selection (character): The mechanism to select parents. Selection mechanisms are "prop","prop_random" or "tournament".

7. nb_groups (int): The number of groups chosen to do using the tournament selection.

8. generation_gap (numeric): The proportion of the individuals to be replaced by offspring.
9. gene_selection (function): The additional selection method for choosing genes in GA. This can be in the form of a user-defined function which takes in two parents and outputs two children.
10. nb_pts (int): The number of points that used in crossover
11. mu (numeric): The mutation rate

Package Structure

The overall package structure is pictured below. The functions with stars indicate they can be modified by user input via the parameters of the select function.



Example

Below are two real examples for fitting. First example, using an R dataset "Boston", with the dependent variable being "crim".

```
library(MASS)
library('GA')
Boston_model1 <- select("crim", Boston)
Boston_model2 <- select("crim", Boston, reg_method = "lm")
Boston_model3 <- select("crim", Boston, interaction = T)
Boston_model4 <- select("crim", Boston, generation_gap = 0.5)
Boston_model5 <- select("crim", Boston, nb_pts = 3)
Boston_model6 <- select("crim", Boston, reg_method = "lm",interaction = T,
                        nb_pts = 3)
```

Second example, using an R dataset "mtcars", with the dependent variable being "mpg".

```
library('GA')
library(datasets)
mtcars6 <- select("mpg", mtcars)
mtcars2 <- select("mpg", mtcars, reg_method = "glm")
mtcars3 <- select("mpg", mtcars, interaction = T)
mtcars4 <- select("mpg", mtcars, mu = 0.4)
mtcars5 <- select("mpg", mtcars, nb_pts = 2)
mtcars6 <- select("mpg", mtcars, reg_method = "glm", interaction = T,
                  nb_pts = 2, mu = 0.4)
```

Test

Unit Test

We did unit tests for most of the auxiliary functions, except those which are really simple. We aimed at testing the correctness of those functions. Those tests are written in Utilities.R. We didn't test the input error because those functions are not allowed used by users. We have full control to assign correct inputs. All tests are passed.

Integration tests

We also did integration tests, which is in test-select.R.

First, we tested the input error. Our funtions successfully threw errors when user provided invalid inputs.

Second, we tested the convergency. We use Boston data. With different mechanism, our algorithm can converge to the same result.

Third, we tested correctness. We used simulated data. According to the data generating mechnism, the result should only include X1, X2, X3. The result is correct.

Member Roles

Skander was the primary coder and bug fixer. He wrote a large proportion of the code and desgined most of the algorithms. Rivers wrote much of the documentation and served as package guru making sure the package had all the necessary components. Yachuan was a secondary coder and wrote most of the tests. Yu was a secondary coder, tester, documenter, and organizer of the group.