STAT243: Final Project

Xiao Li, Louis Rémus, Jinhui Xu, Lei Zhang

Fall 2017

Introduction

This project aims to create a R package to implement a genetic algorithm for variable selection in regression problems, including linear regression and GLMs. By default, AIC is the objective function, but we also allow users to provide their own fitness function. Several genetic operators, like crossover and mutation are available and can be combined in the optimization process. In terms of efficiency, most of the calculations have been vectorized. The result from implementing our package on the real examples are provided. We also carry out formal testings for individual and overall functions.

All codes used in this project can be found in GitHub repository: `https://github.com/lixiao0214/GA`.

Contribution

- *Xiao Li*
  Xiao wrote mutation, replace population and plot part. In addition, he wrote the most part of the final report.
- *Louis Rémus*
  Louis has finished the modelling part. He also provided template for final report and offered support to combine all functions into a main function.
- *Jinhui Xu*
  Jinhui completed algorithm for crossover. He also helped Louis with modelling part and contributed to combine all function together.
- *Lei Zhang*
  Lei worked on how to select parents from the population.

Each team member wrote testing files and documentation for his/her own functions.

Algorithm

To implement the genetic algorithm, we decided to break the complete algorithm into several parts. Each function under each part has its unique role, and we combined all parts together to form a main function, which returns the result after iterations. Aside from the functions, we also developed tests against each function. Those testing files can be found under `tests` and `testthat` folder.

- **Modelling**
  In the modelling functions, our target is to provide goodness of fit scores as defined by the user for every member of the population. Simply put, the output of the modelling section is the population `data.frame`, the `main_dataset` containing the regression target and covariates, and the goodness of fit function chosen by the user, and we have to output for every member of the population the goodness of fit score. To achieve this and for the purpose of clarity, we defined functions as modular as possible. Thus, we defined:

  `get_goodness_of_fit` This function computes the goodness of fit for one regression task, being defined as one `lm` fit and goodness of fit score. It also performs some basic checks to make sure the input is of the appropriate format, and does not lack the `regression_target` or have 0 covariates.

  `compute_population_goodness_of_fit` This function computes the goodness of fit for each element of the given population, repeatedly calling `get_goodness_of_fit` defined above.

- **Select Parents**
  In this part, our target is to select parents to breed based on their goodness of fit. If we directly select parents with probability proportional to the objective function value, we may encounter the problem of dealing with both positive and negative values. So we are going to use only the objective function values' ranks as the fitness. There are three methods. We provide three methods when selecting parents. The first one is to select each parent independently with probability proportional to fitness. The second one is to select one parent with probability proportional to fitness and to select the other parent completely at random. And the third method is the tournament selection. to divide the population into different sections. The best of each section will be selected to form new sections. Such procedures go on until sufficient parents have been generated.

  `selectparents` The input will be a `matrix` containing the population from the current generation and the objective function values of goodness of fit.This function outputs the a `list` containing all the selections of parents. In addition, each element of this list contains a specific pair of couple.

- **Crossover**
  In this part, our target is to get the offspring after we select our parents. Crossover means that both series of genes of parents will be divided into p partitions. One of the offspring will pick one partition for each specific position. And the series of genes of another offspring will be formed by rest partitions. The input should be `list`, and each element of list contains a selected pair of parents. The number of partitions, p, will be determined by the user. Finally, the output will be a `matrix` containing all information of genes of offspring.

  `crossover_p_split` This function outputs the a `matrix` containing all information of genes of offspring. It will cut the series of genes into p partitions and randomly pick the partitions to construct a series of genes for offspring.

- **Mutation**
  In the mutation function, our purpose is to test whether certain cell of an individual mutates or not. The user needs to specify a mutation rate, which is the chance of a cell to mutate. In order to save the computation cost, we used input as `matrix` to do vectorization. The output will be a `matrix` containing the series of genes and corresponding values of goodness of fit.

  `generate_mutation` This function outputs the population with values of goodness of fit after mutation. To complete this task, we generate random numbers from Unif(0,1) to compare with the mutation rate. If the random number is smaller than the mutation rate, mutation will take place. Then by calling `compute_population_goodness_of_fit`, we will get the desired `matrix`.

- **Replace Population**
  In this part, our purpose is to replace population with offspring after mutation. There are two main methods achieving this goal. The first method is to replace the worst proportion of parents with offspring. Another method is to combine parents and offspring together, and then we do not include the worst proportion of them. The input is a `matrix` and output is `data.frame`.

  `get_next_population` This function returns $t + 1^{th}$ generation by different methods of replacing t^{th} generation with offspring.

- **Select**
  In this part, we simply combine functions from previous parts together to form a main function to simulate. We first generate a random set of population. Then we go over the above parts one by one. `max_iter` is the maximum number of iterations we will run. The default value is 100. And if all the individuals in the population are the same, we will stop since the ultimate convergence happens.

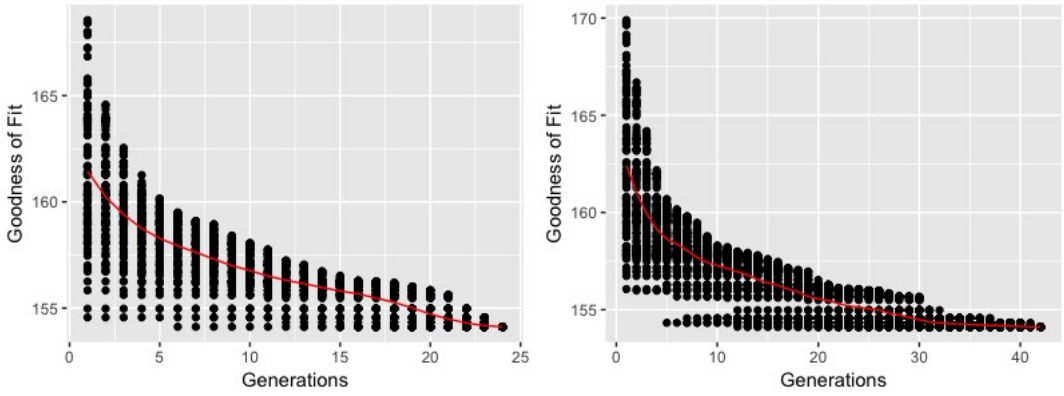
  `select` This function will returns a `list`. The first element of the output is also a `list`, and the sub-element contains all values of goodness of fit for individuals in the order of generations. The second element of the output is a subset of the data we do model fitting. The variables in this subset are the desired variables after variable selection.

Results

Here, we are using `mtcars` in R to test the rate of convergence. We can see that under different conditions, the goodness of fit will converge in the end, but with different rate of convergence. To verify the relationship between the rate of convergence and specific parameter, we held all other parameters constant and changed the input for the testing parameter.

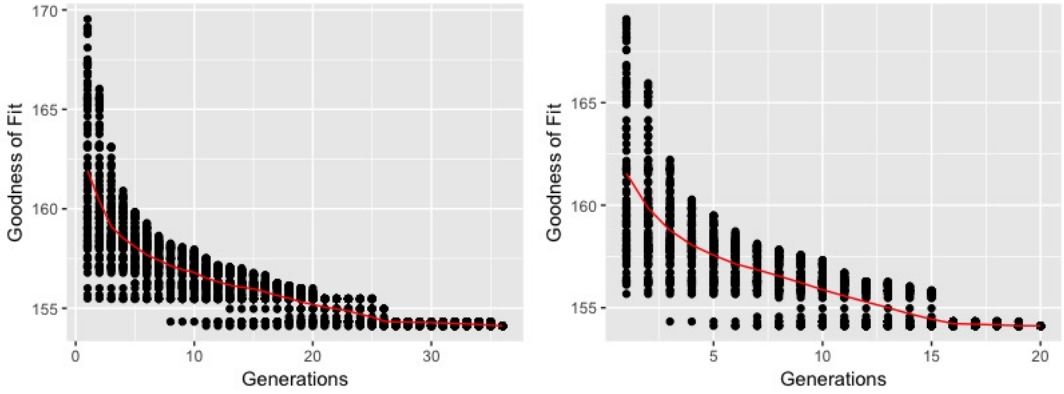
In the plots, the x-axis represents the stage of generations and the y-axis represents goodness of fit. Each point on the graph represents an individual's goodness of fit at t^{th} generation. And the red line represents the average goodness of fit for total population at t^{th} generation.

- **Changing mutation rate**
  Here, we decided to use mutation_rate=0.01 and mutation_rate=0.1 to test the difference in convergence rate. The following images show how goodness of fit for whole population vary with generations.



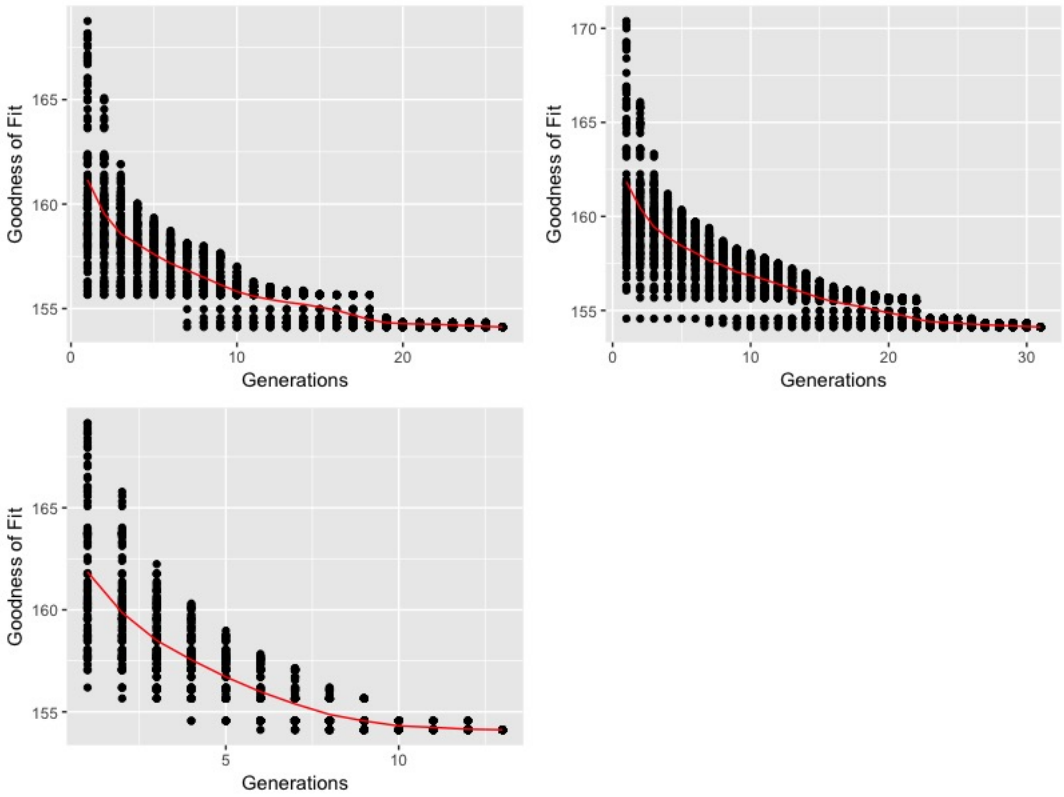
  The left plot shows how goodness of fit changes over time when mutation rate is 0.01. Generally, smaller mutation rate leads to better overall goodness of fit of population. We can clearly see that the smaller the mutation rate, the faster the convergence rate.

- **Changing number of partitions in crossover**
  Here, we decided to use p=2 and p=5 to test the difference in convergence rate. The following images show how goodness of fit for whole population vary with generations.



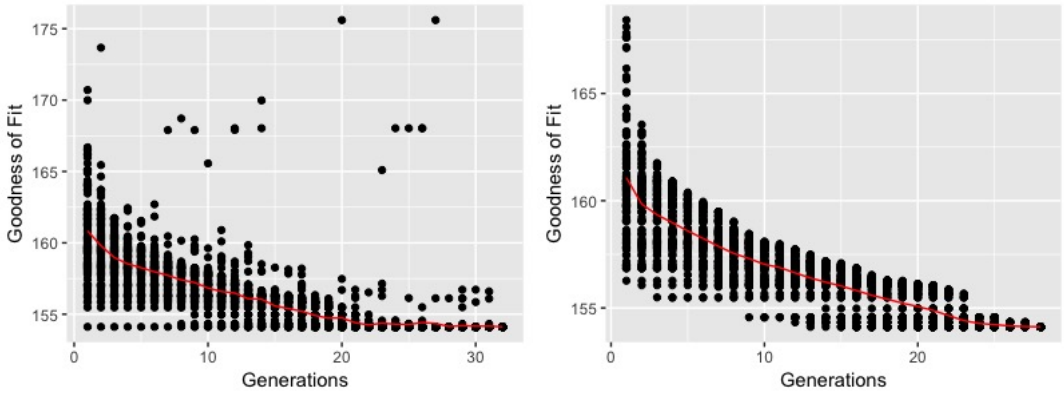
  The left plot shows how goodness of fit changes over time when p=2(number of partitions is 2). We can clearly see that the larger the number of partitions, the faster the convergence rate.

- **Changing methods of selecting parents**
  Here, we decided to use three different methods to test the difference in convergence rate. The following images show how goodness of fit for whole population vary with generations.



The top-left plot shows how goodness of fit changes over time when we select both of parents based on the ranking and the top-right plot shows how goodness of fit changes over time when we only select one of parents based on the ranking. The bottom-left plot is based on tournament selection. The plots indicate that selecting two parents based on ranking has a faster convergence rate than only selecting one parent based on rankings. In addition, the tournament has the fastest convergence rate here.

- **Changing the methods of replacing population**
  Here, we decided to use two different methods of replacing population to test the difference in convergence rate. The following images show how goodness of fit for whole population vary with generations.



The left plot shows how goodness of fit changes over time if we replace the worst proportion of population. And the right plot shows how goodness of fit changes over time if we combine both parents and offspring to select the best ones. The plots have demonstrated that the replacing by proportion seems to have a slower rate of convergence.

Test

Under the `tests` and `testthat` folder, those are the testing files. Within each testing file, we test if the function returns the desired type of output. Also, we test if the dimension and the contents of the output satisfy our requirements. In the end, we have successfully passed all the testing files.

Discussion

By taking all the factors into account, if we want to use this algorithm to do variable selection, choosing smaller mutation rate, larger number of partitions, "tournament" method of selecting parents and "re-rank" method of replacing population will give the optimum rate of convergence. Furthermore, this algorithm is very convenient to use. Users simply offer the dataset and set few parameters. The algorithm will provide users with what they want. If users have different criterion of goodness of fit, they only need to put the name of function in the argument.