# Final Report

Ashlyn Jew, Rhiann Zhang, Tianyue(Steven) Zhou

12/15/2021

## 1 Introduction

In this project, we created a R package implementing genetic algorithms for variable selection in regression problems, including both linear regression and GLMs. We allow users to choose their own fitness functions and by default it's AIC. Several genetic operators, such as crossover and mutation are combined into this whole selection process. We test all of our functions using different datasets and criteria. The results implementing our package on the Boston dataset are shown below.

All code for this project can be found in GitHub repository: https://github.berkeley.edu/asjew/GA

## 2 Algorithms

In order to implement this genetic selection algorithm, we partitioned each step of the algorithm into separate helper functions and formatted our main select function such that it carries out the genetic algorithm by calling each of our helper functions. Because each of our helper functions had a well-defined task, this also allowed each of us to focus on building functions without having to consult each other about any minute changes were being made. Each helper function is briefly described as follows:

### 2.1 Calculate Fitness and Find the Best Model

In the calc_fit function, we aim to obtain fitness values for the population. The user can specify their own fitness function and regression type. For these two, the default are AIC and linear regression. Then we call find_best_model function to pick out the gene which has best fitness score(eg:for AIC, it's the lowest) in the current generation.

### 2.2 Select Parents

In the select_parents function, we aim to select parents on the basis of their fitness scores. We offer two selection methods in this function, one is "prop_rand" and the other one is "tournament". "prop_rand" method chooses one parent with probability proportional to the fitness functions and the other one at random while "tournament" basically chooses the best one in different sections.

### 2.3 Crossover

In the crossover_p_split function, we aim to carry out the crossover operator and obtain offspring. Users are allowed to choose the number of split points. We divide both parents into (p+1) parts. Then we randomly choose whose piece of gene should be inherited by one offspring and the other offspring inherits from the other parent's piece.

### 2.4 Mutation

In the mutate_genes function, we aim to decide whether certain gene would mutate or not. As usual, we let users choose the mutation rate(by default it's 0.01). In the simplest implementation, each gene has an independent probability, mu, of mutating, and the new allele is chosen completely at random from the genetic alphabet.

### 2.5 Select

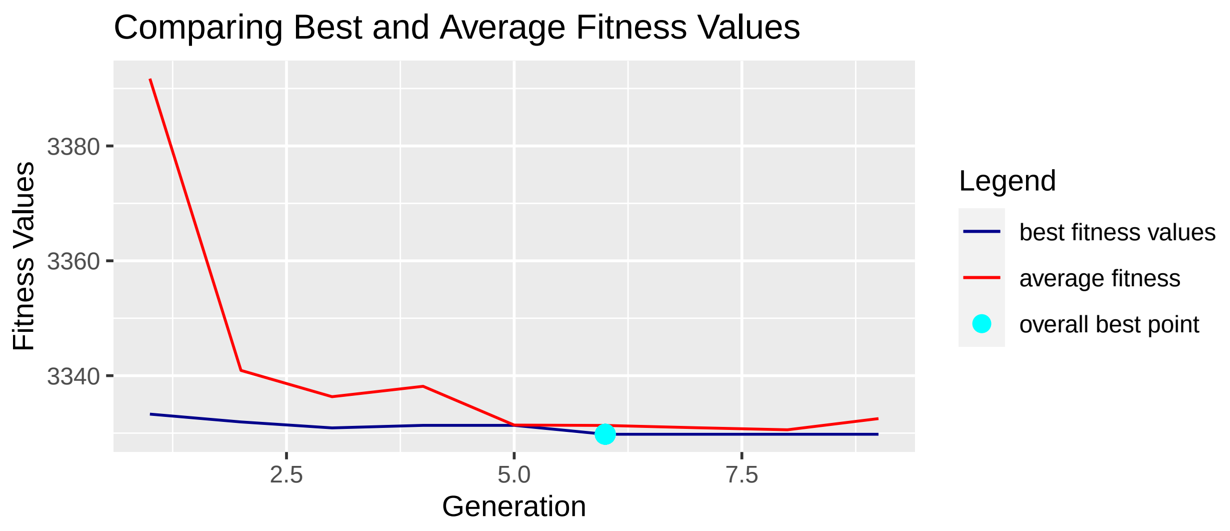
In the select function, we pool previous functions together to form this one. We first call gen_firstgen function to generate the first random generation and then call above functions one by one. The max_iter default is 50 and also can be specified by users. Moreover, we use a scaled relative error between the best fitness score of our current generation and the best fitness score of our previous generation to evaluate diversity. And if this relative error is less than the diversity threshold for three consecutive times, we consider it as convergence. When designing the output of our select function, we decided to only return sufficient information about the best model, such as its fitness score, the generation it was created, as well as the average fitness score of that generation. Additionally, our function also prints a plot that visualizes the changes that occur between the average and best fitness scores throughout the generations of genes and highlights when our best model was created.

## 3 Results

We decided to use the "Boston" dataset from the MASS library to illustrate our results because it is a linear dataset that is commonly used. Under different conditions, it will all converge but with different rates. We thus explore how different methods/parameters affect the rate of convergence.

### 3.1 The default

In this part, we use all the default methods/parameters to select variables. The result is shown as below:

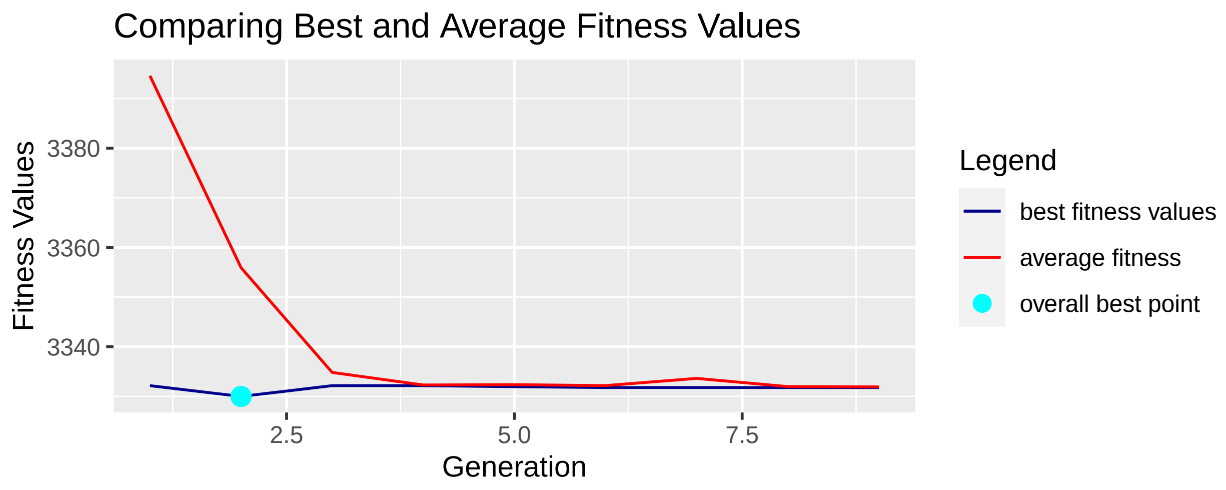


```
##   Generation    Best_Model Best_Fitness_Value Average_Fitness
## 6          6 1001001101111             3329.8        3331.335
##                                                Best_Model_String
## 6 crim ~ zn + nox + dis + rad + ptratio + black + lstat + medv
```

It turns out that using default method for Boston data set will converge very fast, which is usually less than ten generations. Therefore, we obtain our best model pretty quick.

## 3.2 Change the Number of Partitions

Next, we increase our crossover points from default 1 to 6 and see how it changes the convergence rate.

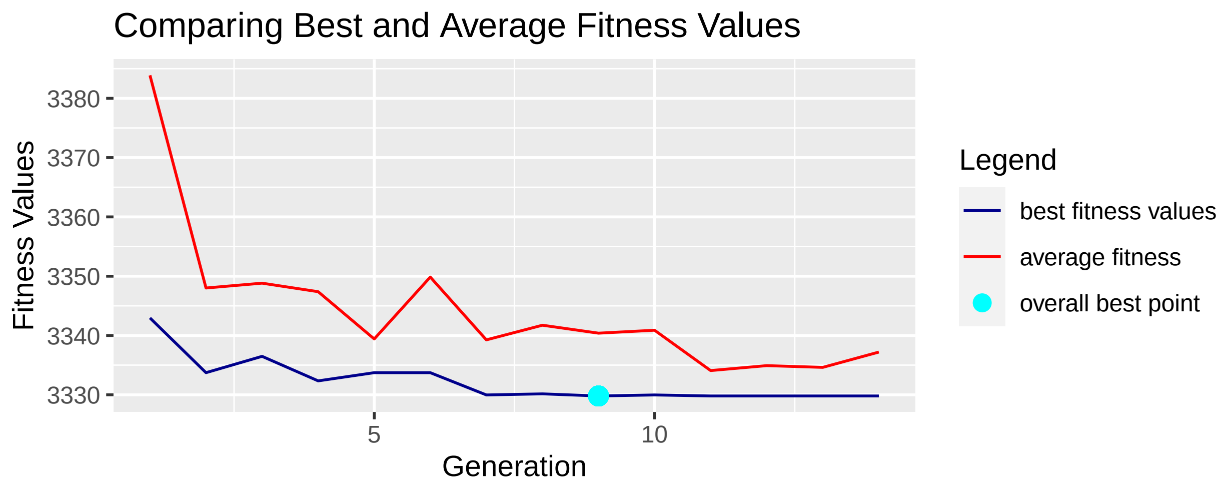


```
##   Generation    Best_Model Best_Fitness_Value Average_Fitness
## 2          2 1101001101111           3329.974        3355.919
##                                                  Best_Model_String
## 2 crim ~ zn + indus + nox + dis + rad + ptratio + black + lstat + medv
```

As shown above, we can easily observe that the more crossover points we have, the faster convergence we achieve.

## 3.3 Change Mutation Rate

We change our mutation from 0.01 to 0.1 and check how it's gonna affect our result.

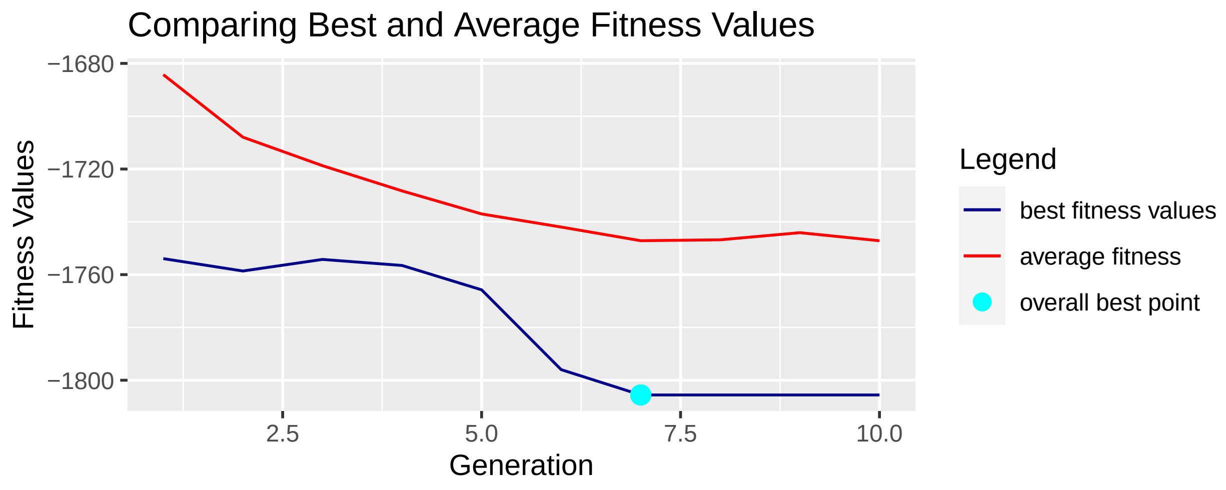


```
##   Generation    Best_Model Best_Fitness_Value Average_Fitness
## 9          9 1001001101111             3329.8         3340.39
##                                              Best_Model_String
## 9 crim ~ zn + nox + dis + rad + ptratio + black + lstat + medv
```

As we expected, generally smaller mutation rate leads to better overall goodness of fit of population. It can be seen that the smaller the mutation rate, the faster the convergence will be.

### 3.4 Change different fitness functions

In the end, we change our objective function to log likelihood and see how well our algorithm performs on this objective.



```
##   Generation    Best_Model Best_Fitness_Value Average_Fitness Best_Model_String
## 7          7 0010000000000          -1805.573       -1747.155       crim ~ chas
```

We can see that it still converges for different fitness functions, which allows users significant customization. They can implement this algorithm based on their specific needs.

## 4 Testing

The testing we carried out was focused on assessing whether our inputs and outputs were consistent with what we were expecting. We also input a large range of values in order to confirm that our functions would not break when dealing with inputs that were extreme, even if they were not practical.
Naturally, we also ensured that our functions did not allow invalid inputs such as string values when it was expecting numeric values, and would return descriptive error messages when doing so.
We also make sure that each helper function was returning an output that was consistent with the next helper function's input.
Additionally, we confirmed that none of our genes were being generated or modified to be completely with zeros and if that did happen, we have simulate_gene function to fix that.

## 5 Contribution

We began our project by brainstorming together and confirming that we all had an cohesive understanding of the general task that we were trying to carry out as well as what smaller actions were expecting from each other. During our meetings, we assigned the smaller tasks amongst ourselves and planned out our next steps together. Ashlyn created the calc_fit.R, test_calc_fit.R, gen_firstgen.R, test_gen_firstgen.R, select_parents.R, test_select_parents.R, and simulate_gene.R files. She also put our package together. Rhiann created the select.R, test_select.R, find_best_model.R, mutate_genes.R files, as well as the tournament section of the select_parents.R file. Steven created the crossover_p_split.R, test_crossover_p_split.R, test_mutate_genes.R. Rhiann and Steven also created the write-up together. And each of us wrote the documentation for each our helper functions.

## 6 Discussion

If users want to implement our package to do variable selection, we recommend to choose larger number of partitions for crossover and small mutation rate. And we do suggest users using "tournament" method for selecting parents. Also they can choose their own fitness function, diversity threshold to deal specific regression problems they are faced with.