On the Design and Implementation of Genetic Algorithms for Multivariate Feature Selection

Andrew Ma

Repository: https://github.berkeley.edu/andrew-c-ma/genetic-algo/

Installation: 'R CMD INSTALL GA'

Abstract

- This paper explores the creation of a genetic algorithm for feature selection in multivariate regression models followed by images and examples of its deployment. While stable methods such as step-wise regression are sure to converge to an optimal solution for feature selection problems, genetic algorithms have an advantage over regression models for models with large numbers of covariates. Step-wise regression algorithms must test every single combination of features in order to arrive at the optimal solution, while genetic algorithms may converge to a candidate solution much quicker with less computational cost, leading large increases in computation efficiency.

Introduction

Our package allows for the implementation of genetic algorithms with the goal of selecting optimal feature combinations resulting in the best fit for generalized regression models.

The structure of our GA package is modular and uses vectorized calculations for efficiency and speed. While largely functional, our program does use some object oriented programming such as passing a user-defined function between various supporting functions, and function results are treated as objects and passed onto other modules. This streamlined approach can be seen in Fig 1. below, where each step produces an object that can be read into the next step of our algorithm.

We chose binary encoding for genes because each linear regression model either does or does not have any given feature. This bit-wise encoding of the genes impacts each element of the genetic algorithm so that crossover and mutation all interact in a bit-wise manner.

Design & Development

Above in figure 1, we show the flow of our algorithm. First we create an initial generation. Next, for each iteration, we score the fitness of the entire population. If elitism is selected, the top proportion of the population is protected from randomized selection, via the Elitism Function. The remainder of the population is selected to be parents based on their fitness, then the genes of those parents are crossed via a crossover process then mutation is applied. We combine the genes then check for termination conditions. If termination conditions are not met we repeat the process. Now we go into a little more detail about our functions features:

Creation of a New Generation

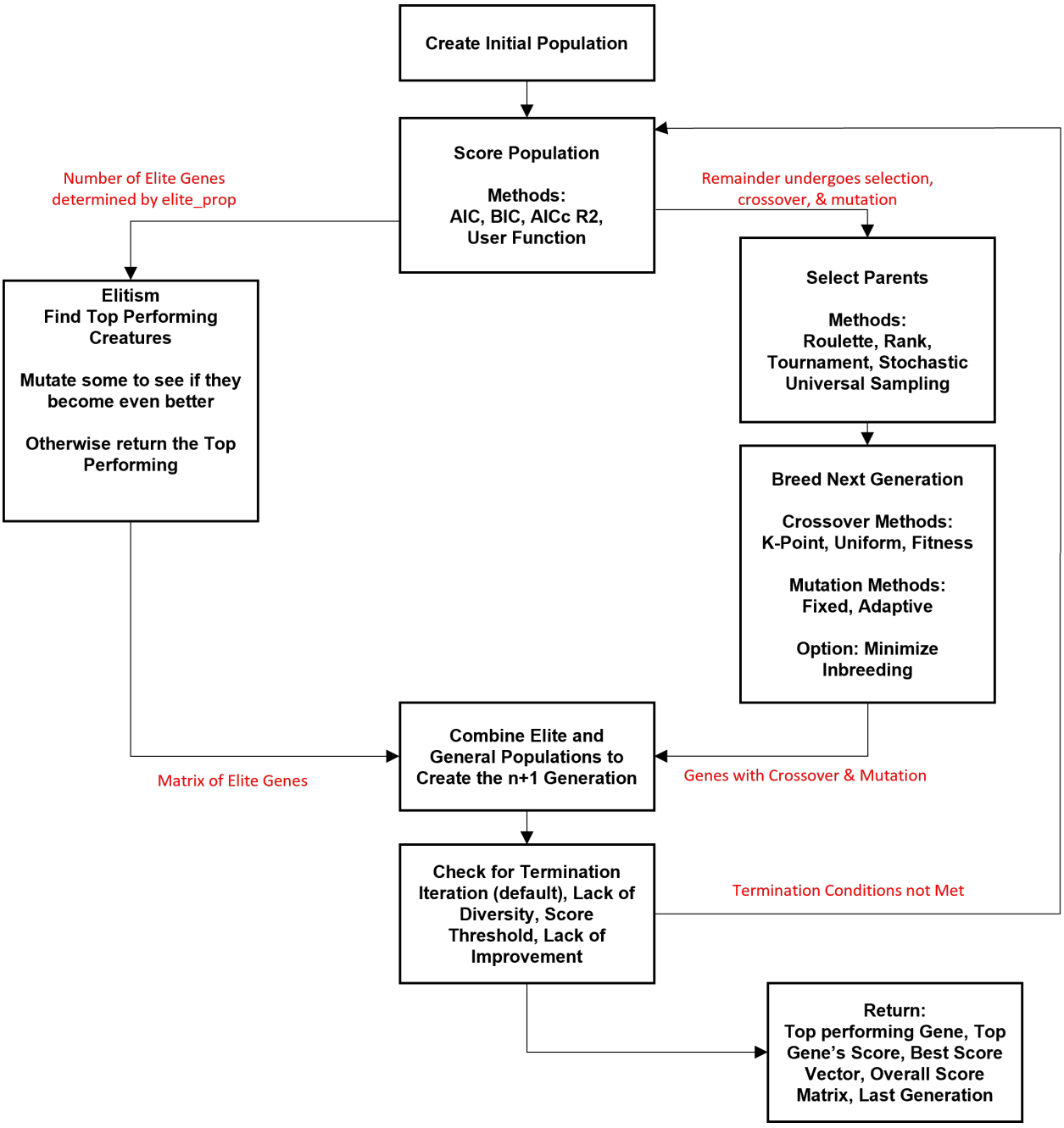

Figure 1: Figure 1. Flowchart of Genetic Algorithm

The function creates a generation matrix with rows representing the individual creatures and column representing the number of genes. As mentioned above each gene can be presence of absence as a 1 or 0. Linear Regression is not able to process the absence of any features so the function checks that all creatures have at least one active gene, and this function gives the option for users to add in their own custom genes for instance starting with a matrix of some known strong performers, with the rest being random still.

Score Fitness: The user can select AIC, R2, BIC, or AICc for evaluation of each generation as well as enter in a custom function.

Custom Fitness Functions: Take a vector representing a single creatures genes and return a score. While there are many ways to do this, the first two arguments must be 'gene' and 'data' and when using the function the user must also specify whether the fittest creature has a high value or low value.

Elitism - We took a unique spin on elitism. Primarily it ensures that the fittest creatures from each generation make it into the next. However we apply a 1 bit mutation randomly to each creature. If the mutation improves the creature, the new creature is returned instead. This speeds up optimization while also improving diversity. Select Parent - Our function allows the user to specify 2 or more parents for each future offspring. The select parents process finds generates parents for each required offspring using four methods. The Roulette method selects parents randomly proportionally to their fitness. This favors very fit parents. The Rank Method selects parents proportional to their rank favoring fit parents but not as steeply. The Stochastic Universal Selection (SUS) method works like roulette but a user specified number of parents are selected at a fixed interval afterwards. Tournament selects n candidates at random and the fittest from that pool becomes a parent. Below in figure 2 we have examples of each using the same fitness scores:

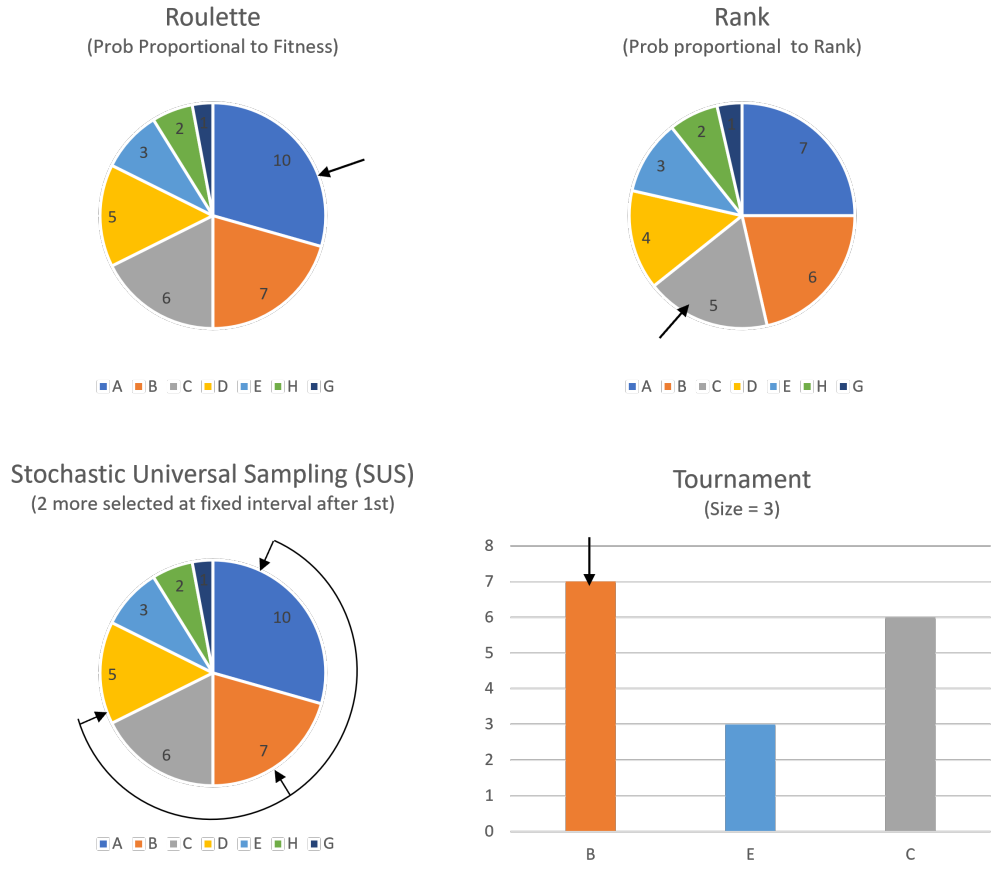

Figure 2: Figure 2. Select Parent Methods

Breed Next Generation:

Next we apply two processes to create the genes for the next generation. First we conduct crossover then we conduct mutation. Additionally, we added a feature to minimize inbreeding. We have three crossover methods: Uniform, Fitness, and K-Point. Uniform Crossover selects each individual gene of the next offspring one at a time assigning it from a parent randomly at a proportion equal to the number of parents. Fitness generates each individual gene randomly from a parent proportionally to the fitness of each parent. K-Point crossover breaks the gene at k points, and randomly assigns each point to one of the parents, the offspring being the combination of those segments. k=1 is a 1-point crossover. See Figure 3.

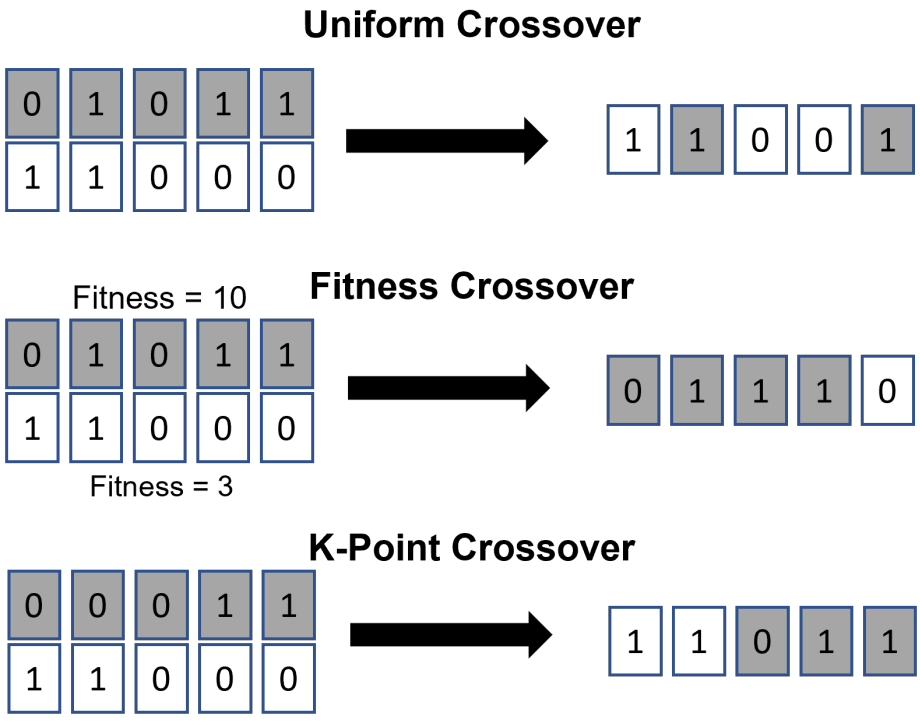

Figure 3: Figure 3. Mutation Types

Next mutation is applied either at a fixed rate, or adaptively. Fixed mutation selects offspring at a random fixed rate. Next the function applies a single bit change to a random gene. Adaptive Mutation follows the same process but the mutation rate changes based on the diversity of the population (See figure 4). The function allows the user to select a maximum mutation rate, a minimum mutation rate and an inflection point to generate a logistic function of the populations diversity.

Minimizing inbreeding:

This option re-arranges parents a lower the chance of similar genes breeding. It does this by selecting new patterns (from the already selected parent list) proportional to the square of the difference of their genes. Once rearranged the function proceeds as above.

Termination Conditions:

After genes are recombined, the function checks if termination conditions are met. The Default is iterations, and the user is able to select a combination of three additional termination conditions. Reaching a fitness score threshold for instance AIC < 500. Gene Diversity fouling below a certain threshold for

instance less than 10% unique genes remaining and pause length, for instance minimum AIC has not changed in 20 generations. In each case the user selects the conditions along with the governing parameter (Min, Max, Mean, Median).

Function Output:

Upon reaching termination conditions, the function returns a list with the following items: the most fit gene sequence, that gene's score, the best scores by generation, the data frame of scores by generation, and the final generation of genes for the entire population.

Testing & Examples

Here we do three examples for our function. The first two are from select()'s help function (?select). In the third example we use a much large data sample from generate_data() and implement the adaptive mutation and minimize inbreeding. Immediately below is an example of the code being ran. For conciseness we load the functions using source('DEMO.R') and the code can be reviewed in our github.

```
out3 <- select(
  total_number_generations=total_number_generations,
  number_of_parents = number_of_parents,
  pop=pop,
  gene_length=gene_length,
  prob = prob,
  user_genes = NULL,
  response_vec = response_vec,
  independent_vars =independent_vars,
  method = method,
  tourn_size = 4,
  mutation = mutation,
  mutation_rate = mutation_rate,
  minimize_inbreeding = minimize_inbreeding,
  crossover = crossover,
  elitism = TRUE,
  elite_prop = elite_prop,
  ad_max_mutate = ad_max_mutate,
  ad_min_mutate = ad_min_mutate,
  ad_inflection = ad_inflection,
  ad_curve = ad_curve,
  estimator = estimator,
  pause_length = pause_length,
  percent_converge = percent_converge)

plot(out3[[3]],xlab="generation",ylab="AIC, Most Fit Creature")
```

```
library(GA)
source('DEMO.R')
```

We did a lot of initial research into different methods for genetic algorithms. At an cursory level, any genetic algorithm design has to balance speed of convergence and computational complexity. The code for this final example is available in 'Final_Demo.R' but since it is time consuming and lengthy in code length we chose not to knit. In this last example we compare three methods

- Method 1: Fast - two parents, 1-point crossover, high elitism, high mutation

- Method 2: Multi-Parent - four parents, lower elitism, lower mutation, uniform crossover
- Method 3: Adaptive - two parents lower elitism, adaptive mutation, minimize inbreeding.

For this experiment, we use the data from generate_data() which is designed in such a way that the coeffieients from features decrease from 1 to 50. See ?generate_data. We run each method five times taking the average system.time and also scoring the SSE using the formula below:

```
test_out_results<- function(list_o) {
  iter <- length(list_o)
  final <- iter
  b<-c(rep(1,25),rep(0,25))

    for (i in 1:iter) {
      if (is.vector(list_o[[i]])) {
        a <- list_o[[1]]

        } else {
          a <- list_o[[i]][1,]
          }
      final[i] <-  sum(abs(a-b))^2
    }
  return(mean(final))
  }
```

Here in figure 5 and 6 we print then plot our data. We see that maintaining diversity had better results (minimizing SSE). However we also see that elitism appears to be computationally wasteful. If we went back and added additional features we could make the mutation in elitism optional which should speed up the processes.

```
    time SSE
1 24.664 484
2 24.254 225
3 17.088  81
```

Figure 4: Figure 5. SSE and Time

References:

1. Eiben, A. & Rau, P-e & Ruttkay, Zsófia. (1997). Genetic Algorithms With Multi-Parent Recombination.
2. Givens, G. H., & Hoeting, J. A. (2012). Computational Statistics: Givens/computational statistics (2nd ed.). Wiley-Blackwell.
3. Hassanat, A., Almohammadi, K., Alkafaween, E., Abunawas, E., Hammouri, A., & Prasath, V. B. S. (2019). Choosing mutation and crossover ratios for genetic algorithms—A review with a new dynamic approach. Information (Basel), 10(12), 390. https://doi.org/10.3390/info10120390
4. Katoch, S., Chauhan, S. S., & Kumar, V. (2020). A review on genetic algorithms: past, present, and future. Multimedia Tools and Applications, 80(5), 1–36. https://doi.org/10.1007/s11042-020-10139-6
5. Trejos, Javier & Villalobos-Arias, Mario & Espinoza, Jose. (2016). Variable Selection in Multiple Linear Regression Using a Genetic Algorithm. 10.4018/978-1-4666-9644-0.ch005.
6. Umbarkar, Dr. Anantkumar & Sheth, P.. (2015). CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW. ICTACT Journal on Soft Computing (Volume: 6 , Issue: 1). 6. 10.21917/ijsc.2015.0150.

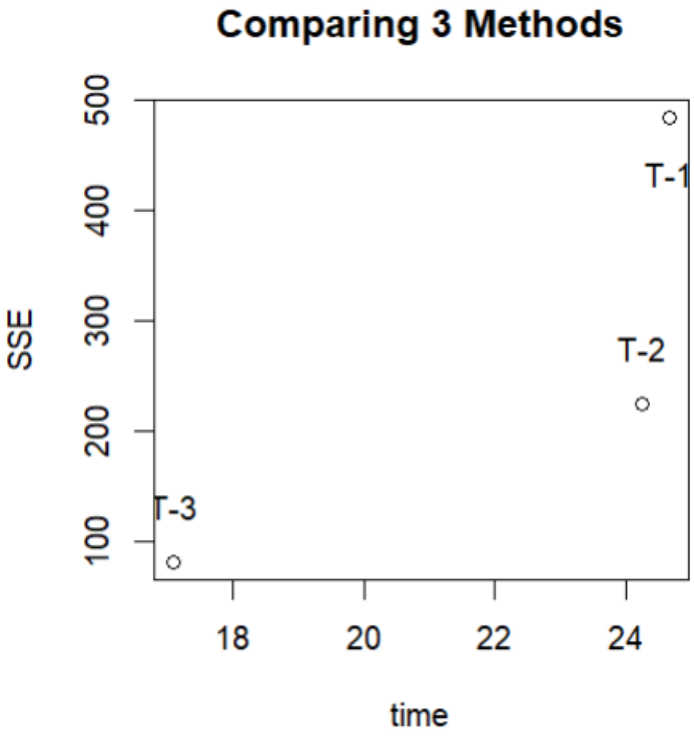

Figure 5: Figure 6. Comparison of Methods