Final Project Report

Fernando Machado, Jiayang Nie, Kathrin Lorenz

12/14/2021

Github

All the codes are under "jnieser/GA" in Github. Also, testthat::test_package('GA') will take approximately 30 seconds to run.

Description of the GA Algorithm

1. Brief description of the basic idea of the GA algorithm behind the package

The GA Package is used to obtain the optimal groups of variables for linear regressions of a given data set, by default according to the Akaike Information Criterion, an estimator used to measure the relative quality of statistical models given a data set. User is also allowed to give their own criterias to assess a model. Note, the user-given criteria must also be optimized when the criteria value is minimized.

The main idea is that the user can input a data set and obtain a set of regressions over a selected dependent variable which, if included in the model, make the overall regression have a (close to) minimum AIC value, or equivalent optimal fitness metric. These model (or equivalently the selections of coefficients that make them up) are said to be fit once a low enough AIC is observed.

If we let a linear regression over a data set containing n candidates for independent variables:

$$Y = \beta_1 \cdot X_1 + \beta_2 \cdot X_2 +, ..., +\beta_n \cdot X_n$$

then one gene is defined by its chromosome composed of 1s and 0s where a 1 in the i-th position indicates that the variable is taken in the regression with its corresponding coefficient.

The algorithm thus takes in a population of possible regressions, selects the most fit according to the value criterion with either tournament or rank-based selection (the user chooses the method), crosses the population over, and mutates a portion of this output. The whole process iterates a number of times until the resulting population of regressions scores an appropriate AIC score (or other optimal fitness value), or the stop criteria for the iteration is met.

Details on the modules and OOP justifications

2.A.1 Evaluation of the population

The evaluation is the first stage of each iteration. It assigned a fitness value to each gene in the population, to be used by selection in the next module. As we previously mentioned, one of the user inputs is the function that establishes the fitness metric criterion, which defaults to the AIC method. The remaining inputs should be clear from the previous description: an m by n matrix of the dependent variables, the isolated independent variable Y to be regressed on, and the data set itself composed of all the chromosomes.

For the genes whose chromosomes are composed solely of 0-valued alleles, the worst fitness possible (positive Inf, remember that we seek to minimize the score of whatever input is decided by the user) is assigned, since these do not even represent regressions. For each gene in the population, a glm model (where use of the stats package was made) is constructed based on the chromosome, and evaluated according to the fitness criteria, and the type of regression assigned by the user.

The final output should be a new list containing the genes and their corresponding fitness score.

2.A.2 Hall-of-Fame Genes

In this module, we also keep track of the highest performing candidates as we make iterative progress. A Hall of Fame is essentially a subset of the whole population with the highest scoring individuals for each iteration. It is initially set to a null matrix, then if the fitness score of an gene assigned by the evaluation method described so far as we loop through the population outperforms worst gene in the current Hall of Fame, then the latter is removed and the former is included in the Hall of Fame. This subset of the population of high performing candidates is part of the evaluation output, but not utilized in the subsequent models. The Hall of Fame of the last iteration is indeed the final output of the algorithm.

2.B. Selection methods: choosing the most fit parents

The user can choose between rank-based and tournament selection, with the latter being the default.

The input of the function selection (the only function in this module) will be the output of the evaluation code described above. It selects the best P pairs of genes as parents in preparation for the crossover. The selected parents will be stored as their corresponding column indexes of the genes matrix. The final output will be a 2 by P matrix with each column containing the parents' index in the original matrix.

The function defaults to tournament selection. This method extracts random groups (randomized with the sample function with replacement) by generating a matrix 6 times the size of the population (since partitions will be made for each of the 2 parents by groups of 3). The resulting matrix is therefore ran by the function tournament_vectorization, applied to each element, whereby groups of size 3 are selected for each parent. Out of each of these groups of 3, the gene with the minimum fit value of said group "wins the tournament" and becomes the selected parent.

Should the user input specify for rank-based selection instead, each gene corresponding will be chosen with a probability of 2 * rank / (P * (P + 1)), where rank is the rank of such gene in the population. The best chromosome has probability $2/(P + 1)$ of being selected, roughly double that for the median. This probability is assigned as a function of the fitness rank, where the base function rank that returns the sample ranks of the values in the population is used to obtain the ranks for the probability assignation. Lastly, the pairs of selected parent ranks are replicated in accordance with this rank-based probability, so ultimately our output will be the same as with tournament selection, a 2 by P matrix with each column containing the parents' index in the original matrix.

2.C. Crossover: breeding new genes from best fit parents' genes

Crossover seeks to get a new gene formed by the juxtaposition of a left truncation of parent A with the right tranche of parent B. The selected parents in the previous module is thus the input for the function, since we aim to cross the most fit parents' genes over with one another.

The cutoff point is the allele that determines the last position of parent A's tranche (so parent B's initial allele in the new gene is given by the position next to the cutoff point). This point is decided at random for each gene, and uniformly so (each allele has the same probability of being selected as the divisor of the two tranches).

A crossover_helper function segregated from the main function does the actual crossover for each pair of parents, and is called by the main function with apply over the output of the previous module, which should give us (hopefully more viable) offspring for the current iterations' selected parents.

2.D. Mutation

Mutation alters the genome of the population for a small subset of the population. For said subset, some alleles are changed by switching value (since our alphabet is binary '{0,1}', our case for mutation is simple). Each gene has the same probability of being picked for mutation, called the mutation rate.

The mutation rate can be defined by the user, but it will default to the inverse of the number of alleles in the populations' genes' chromosomes if no probability is specified. The probability is assigned to a binomial distribution which indexes the mutation candidates (or mutants) in the population of crossed over genes taken from the previous module.

Only one allele is selected for mutation for each mutant, uniformly so (we used the sample function to select the position of the allele by shuffling all the numbers up to the length of the chromosome and picking the first value in the sample). This is done separately by the random_mutate function (the switch is done using the absolute value function, so if our selected allele is 0, it switches to the absolute value of -1).

2.E. Iterations

The above modules will be ran repetatively until either a user-defined max-Iter number is reached or the model seems to converge. The converging criteria is determined by whether the ratio of difference of the mean fitness value between two consecutive generations is lower than 0.01.

The final output will be the last generation and the Hall-of-Fame that contains all the all-time-best genes. The reason for providing Hall-of-Fame is that some high-performing gene might be discarded in some selection process due to chance.

Disclosure of each member's contributions:

Fernando was responsible for coding the modules for mutation, crossover, and preparing the report. He also reviewed the entire code for typos or possible misnomers that could impact code style or presentation.

Jiayang covered initialization, evaluation, selection, and setting up the package. Given his very high level of proficiency and previous experience with genetic algorithms, he also helped review Fernando's parts for mutation and crossover and improved the code / helped with turning the ideas into functional code that stream well with the rest of the modules.

Kathrin helped in mapping out the overall structure of the package. She was responsible for the assertions in the select module. She was also responsible for writing tests.

Illustration of the model

Here, two random datasets of X are given, and Y is a perfect linear combination of the first dataset. In this example, we tried to fit a GA model to find the best covariates to be regressed on.

```
#path_to_file = "C:\\Users\\86773\\Documents\\GitHub\\GeneticAlgorithm\\GA_0.0.1.1.tar.gz"
#install.packages(path_to_file, repos = NULL, type="source")
library(GA)
set.seed(5)
X = matrix(rnorm(120*100, 10, 5), 120, 100)
y = rowSums(X)
X_noise = matrix(rnorm(120*100, -10, 1), 120, 100)
X = cbind(X, X_noise)
```

Clearly, the fitness value is decreasing properly.

```
test1 <- select(X,y, max_iter = 100, verbose=2)
```

```
## current iteration:  0
## mean of current population fit:  1074.24
## current iteration:  1
## mean of current population fit:  915.8836
## current iteration:  2
## mean of current population fit:  668.3218
## current iteration:  3
## mean of current population fit:  -338.3768
## current iteration:  4
## mean of current population fit:  -849.7957
## current iteration:  5
## mean of current population fit:  -1506.501
## current iteration:  6
## mean of current population fit:  -2056.249
## current iteration:  7
## mean of current population fit:  -3141.424
## current iteration:  8
## mean of current population fit:  -3505.509
## current iteration:  9
## mean of current population fit:  -4329.468
## current iteration:  10
## mean of current population fit:  -4136.773
```

```
## current iteration:  11
## mean of current population fit:  -4567.233
## current iteration:  12
## mean of current population fit:  -4664.8
## current iteration:  13
## mean of current population fit:  -4670.605
```

```
test2 <- select(X,y, max_iter = 100, select.type="rank", verbose=2)
```

```
## current iteration:  0
## mean of current population fit:  935.393
## current iteration:  1
## mean of current population fit:  840.5156
## current iteration:  2
## mean of current population fit:  394.0195
## current iteration:  3
## mean of current population fit:  -64.04119
## current iteration:  4
## mean of current population fit:  -1307.301
## current iteration:  5
## mean of current population fit:  -2110.934
## current iteration:  6
## mean of current population fit:  -3031.42
## current iteration:  7
## mean of current population fit:  -3863.196
## current iteration:  8
## mean of current population fit:  -4850.534
## current iteration:  9
## mean of current population fit:  -5188.556
## current iteration:  10
## mean of current population fit:  -5378.531
## current iteration:  11
## mean of current population fit:  -5567.623
## current iteration:  12
## mean of current population fit:  -5626.61
## current iteration:  13
## mean of current population fit:  -5759.027
## current iteration:  14
## mean of current population fit:  -5571.729
## current iteration:  15
## mean of current population fit:  -5803.526
## current iteration:  16
## mean of current population fit:  -5712.515
## current iteration:  17
## mean of current population fit:  -5616.847
## current iteration:  18
## mean of current population fit:  -5728.345
## current iteration:  19
## mean of current population fit:  -5879.723
## current iteration:  20
## mean of current population fit:  -5896.273
```

```
test1$bestModel
```

```
##   [1] 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 1
##  [38] 1 1 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1 1 1 0 1 1 0 1 1 0 0 1 1 1 1 1 0 0 1 1 1
##  [75] 0 0 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 0 1 0 1 1 0 0
## [112] 1 1 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 0 1 0 0 0 0 1 0 0 1 1 1 1 0 1 1 1 0
## [149] 0 1 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 0 0 1 1 0 1 1 1 0 0 1 0 1 1 0 0 1 1 0
## [186] 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1
```