STAT 243 Final Project: Genetic Algorithm for Model Selection

Chris Gagne,Zicheng Huang,Yuchen Zhang,Mengying Yang

Dec 15 2017

Description of Algorithm

Our package has one main function `select()`. This applies a genetic algorithm to select the best linear model in a dataset, which contains rows as observations and columns as potential variables.

We modularized our code by breaking down the genetic algorithm into these steps and corresponding functions:

- (1) generating the initial population of chromosomes `initialize()`,
- (2) calculating the fitness for that population `fitness()`,
- (3) selecting parents for breeding `selection()`,
- (4) breeding parents and choosing the next generation `nextGeneration()`. Within this function, we use `crossover()` as the genetic operator to do the actual breeding which includes doing crossover of the parent chromosomes to obtain offspring and added a small amount of randomness into the chromosome using `mutation()`.

Modification of Algorithm by User Input

We also allow user input to adjust the genetic algorithm in a number of ways:

We implemented 3 ways to select parents for breeding. The first method uses fitness-rank-based probabilities to select all parents. The second method uses fitness-rank-based probabilities to choose one parent in each pair, and chooses the second parent randomly. We found that this method does better at preserving diversity in population, thus preventing early convergence. The third method uses tournament selection. This method splits the population into K groups, selects the fittest parent in each group and repeats the process until enough parents have been selected for breeding.

We also implemented a flexible generation gap – the percentage of children replacing the parents in each generation. This had a large impact on the convergence of the algorithm to the global optimum (see examples below).

We also allow the user to specify the number of generations, the starting population size, to use glm vs. lm, and the fitness function.

Variable Representation

We chose to represent our chromosomes as boolean vectors, which are used to index the data matrix. The population set of chromosomes is a matrix of these boolean chromosomes and is passed into various functions. `nextGeneration()` returns a modified version of the population set of chromosomes. We did this, rather than keep old generation chromosomes, to save memory. We do, however, store the fitness scores for each model in each generation that we can plot (see examples).

Installation

Our package can be installed by either downloading and installing, or installing directly from github.

```
install('GA')
install_git('RPackageGroupProject/GA')
```

The package is kept in group folder: https://github.com/RPackageGroupProject/GA
The owner of the repo is: mindyyang

Documentation

The documentation can be found by using the following:

```
?GA::select
?GA::crossover
```

Note that only the `select()` and `plotGA()` are directly accessible to users.

Tests

The package can be tested using the 'testthat' package.

```
library('testthat')
library('GA')
test_package('GA')
```

Example 1: basic usage

Our package comes with many defaults, so at the minimum it can be used in one line using `select()`. All it requires is a dataset, where the first column is the predictor (y) and the rest of the columns are potential variables for the variable selection.

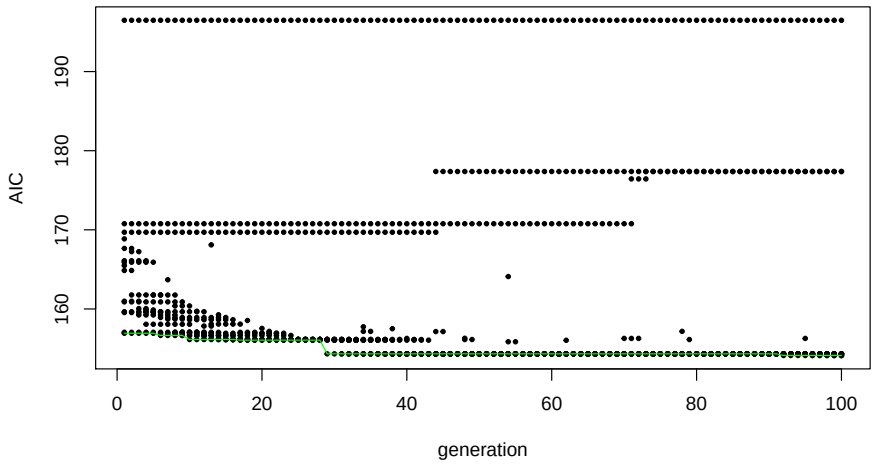
```
library('GA')
results<-select(mtcars)

## [1] "Generation:  10  Fitness:  156.134840745305"
## [1] "Generation:  20  Fitness:  156.058367373584"
## [1] "Generation:  30  Fitness:  154.32736860132"
## [1] "Generation:  40  Fitness:  154.32736860132"
## [1] "Generation:  50  Fitness:  154.32736860132"
## [1] "Generation:  60  Fitness:  154.32736860132"
## [1] "Generation:  70  Fitness:  154.32736860132"
## [1] "Generation:  80  Fitness:  154.32736860132"
## [1] "Generation:  90  Fitness:  154.32736860132"
## [1] "Generation:  100  Fitness:  154.119370868901"
## [1] "ALgorithm Ends"
```

Our function returns the best fitting model.

```
results$fittestModel

##
## Call:
## model(formula = form, data = dat)
##
## Coefficients:
## (Intercept)           wt         qsec           am
##       9.618       -3.917        1.226        2.936
```

It also returns fitness scores for all generations, which can be plot using the our built in function `plotGA()`. The points are the fitness for individual models, and the line is the best fitness achieved at each generation.

```
plotGA(results)
```



Example 2: comparison against backwards selection

We compared the results of our model to backwards selection. Backwards selection chooses the model chooses the same model containing regressors for (wt, qseq, am). This seems to be the best model, as it's the lowest we've seen in any iteration of our algorithm. We also tried forward selection, which does not find this model.

```
full=lm(mtcars[,1]~., data=mtcars[,-1])
step(full, direction="backward",trace=FALSE)


##
## Call:
## lm(formula = mtcars[, 1] ~ wt + qsec + am, data = mtcars[, -1])
##
## Coefficients:
## (Intercept)           wt         qsec           am
##       9.618       -3.917        1.226        2.936
```

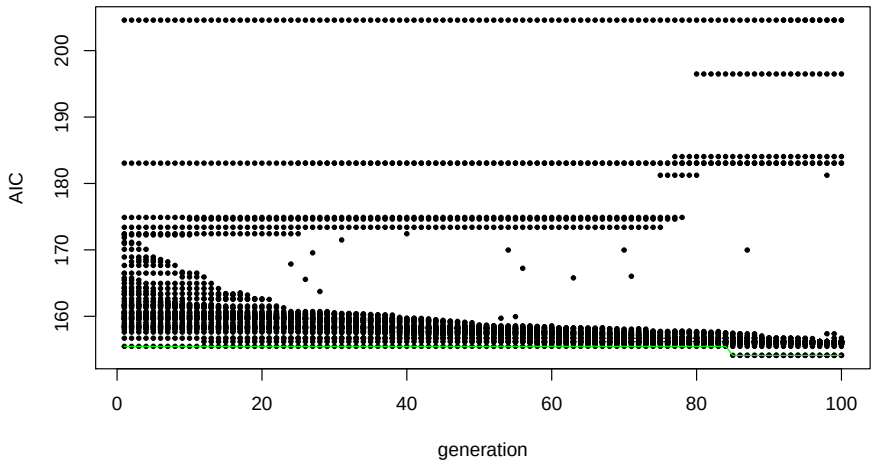
Example 3: user inputs

Users can specify a number of different parameters.

```
library('stats')
userfunc<-function(fit,...){return(extractAIC(fit,...)[2])}
results<-select(dat=mtcars, # dataset
                P=50, # population size
                model=glm, # model lm or glm()
                numGens=100, # number of generations
                G=0.25, # generation gap
                method=2, # parent selection method
                K=2, # number of groups for tournament selection
                fitnessFunction=userfunc, #user defined fitness
                verbose=FALSE, # whether to print results
                )

## [1] "ALgorithm Ends"

results$fittestModel

##
## Call:  model(formula = form, data = dat)
##
## Coefficients:
## (Intercept)           wt         qsec           am
##       9.618       -3.917        1.226        2.936
##
## Degrees of Freedom: 31 Total (i.e. Null);  28 Residual
## Null Deviance:     1126
## Residual Deviance: 169.3  AIC: 154.1
```

We can plot this as well. Notice that with a smaller generation gap, the population maintains variance for longer.

```
plotGA(results)
```



Example 4: early convergence

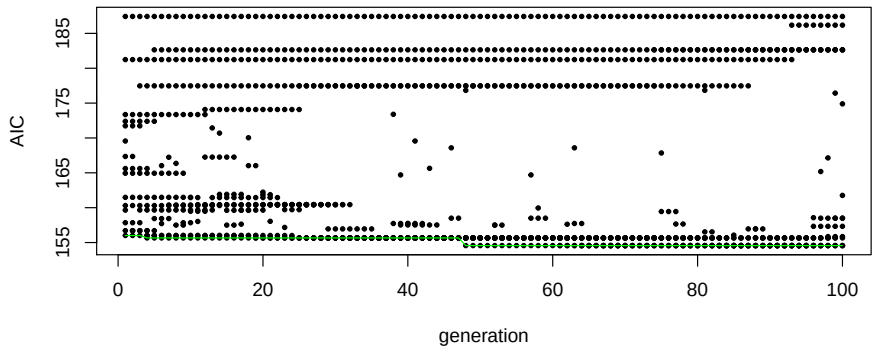
A big difference in our algorithm is the generation Gap. With too high of a generation gap, the algorithm often converges to a local minimum. We considered adjusting the mutation rate and not allowing duplicate parents. These should both help.

```
results<-select(dat=mtcars,G=.8,verbose=TRUE)

## [1] "Generation:  10  Fitness:  155.657784908588"
## [1] "Generation:  20  Fitness:  155.657784908588"
## [1] "Generation:  30  Fitness:  155.657784908588"
## [1] "Generation:  40  Fitness:  155.657784908588"
## [1] "Generation:  50  Fitness:  154.563104963084"
## [1] "Generation:  60  Fitness:  154.563104963084"
## [1] "Generation:  70  Fitness:  154.563104963084"
## [1] "Generation:  80  Fitness:  154.563104963084"
## [1] "Generation:  90  Fitness:  154.563104963084"
## [1] "Generation:  100  Fitness:  154.563104963084"
## [1] "ALgorithm Ends"

results$fittestModel

##
## Call:
```

```
## model(formula = form, data = dat)
##
## Coefficients:
## (Intercept)           wt         qsec           am         carb
##     12.8972      -3.4343       1.0191       3.5114      -0.4886
```

You can see that most of the population converged to the local minimum.

```
plotGA(results)
```



Contributions of Team Members

The project consisted of the following components. We put who contributed to each part in parentheses:

- design of algorithm (all members)
- select function (Chris and Zicheng)
- parent selection (Yuchen and Mengying)
- crossover (Yuchen and Mengying)
- other function (all members)
- package organization (Chris)
- tests (all members)
- pdf document (Chris and Zicheng)