Stat 243 Class Project: Building a Genetic Algorithm Based Variable Selection Algorithm

Joy Hou, Kevin Li, Greta Olesen

December 10, 2014

1 Introduction

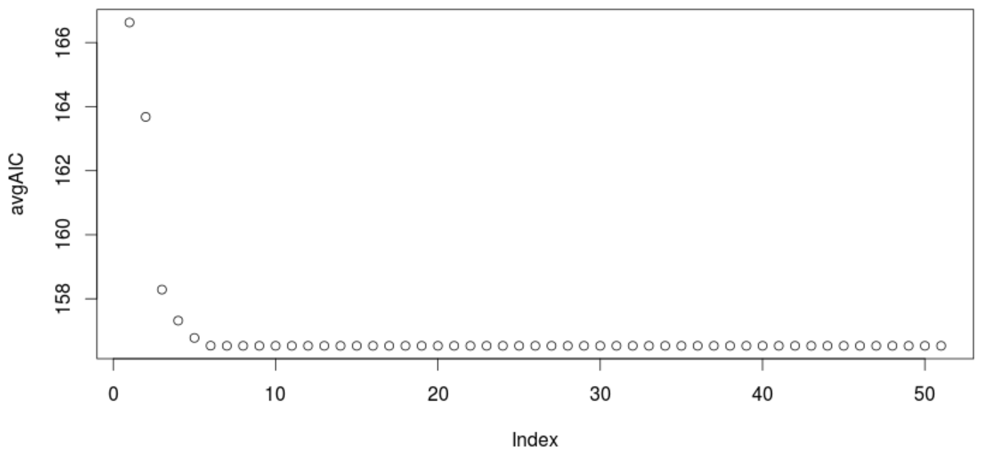
Genetic Algorithms are search heursitics that mimic the process of Darwinian natural selection. As Givens and Hoeting (2005) states, candidate solutions to a maximization/minimization problem are considered as biological organisms represented by the genetic code which specifies the model attributes. Genetic algorithms are especially useful in large scale combinatorial and nonlinear optimization when traditional optimization techniques become untractable. Rather than exhaustively searching for the global optimal solution, the Genetic Algorithm utilizes heuristic-based search methods and returns a solution close to the global optimal solution.

Variable selecton problems, when presented with a large set of potential predictors, become increasingly computationally expensive. As a result, practitioners and academics have turned to search heuristics such as the Genetic Algorithm.

The basic genetic algorithm is as follows:

1. Initialize the first generation of models by generating a random population of n chromosomes
2. Evaluate the fitness/performance f(m) of each chromosome/model m in the first generation
3. Create a new population by repeating the following steps until the termination condition is met:
   (a) Select n/2 pairs of parents chromosomes/models from the previous generation according to the fitness (fitter chromosomes have a greater chance to be selected)
   (b) Crossover is carried out with a crossover probability. If crossover was performed, the 2 parent chromosomes are crossed over to produce 2 children. If no crossover was performed, 2 copies of the original chromosomes/models will be kept.
   (c) Mutation is carried out with a mutation probability. When mutation is carried out, the new offspring is mutated at each locus.
   (d) Accept the children/offsprings as the next population
4. If the terminal condition is satisfied, stop, and return the best solution in the current population. Otherwise, return to step 2

The genetic algorithm by no means returns the optimal solution, but users can generally expect an acceptable solution with rapid convergence that often resembles that presented figure below:



2 Code Structure

We utilized functions in R to carry out our Genetic Algorithm. We chose to make use of functions rather than OOP methods because the functions could be put to use in a clear, orderly manner. Consequentially, the algorithm is outlined as follows:

```
result <- select(X, y, popSize, criterion, type, family, criFun, max_iterations,
                 crossRate, mRate, zeroToOneRatio)
```

Please refer to the help manual for more information about each argument of *select*. The function *select* employs all of the auxiliary functions that are required to carry out the Genetic Algorithm.

The following is a list of brief descriptions of the main auxiliary functions that are utilized in the primary *select* function.

popInitialize function

This function randomly generates the initial population to start the Genetic Algorithm. Its main arguments are the desired population size and appropriate gene length (the number of potential predictors).

The function returns a matrix with the following dimension: $popSize \times geneLength$. Each row of the matrix represents an initial parent/model. Each column of the matrix represents a variable that could potential be included in the final model.

The function makes sure to check that each individual includes at least one variable as a predictor. If the individual does not include any predictors, it is omitted and regenerated until it fits the criteria.

evalFunction function

The *evalFunction* makes use of parallelization to evaluate each individual/model in the current population based on some criterion.

This is a function called *singleEval* that is implemented inside *evalFunction*. *singleEval* evaluates a single model and returns the criterion value for that model. The criterion can be one of the built-in criterions: AIC or BIC. The user can also input their own function into the argument *criFun* to evaluate the model.

The user inputed *criFun* is the function that should be minimized. If specified, the user's function should take in an lm object and return a single criterion value. The following is an example of a function that the user could pass as an argument for *criFun*:

```r
fun <- function(lm_ob){
    adj_r_squared <- summary(lm_ob)$adj.r.squared
    return(-adj_r_squared)
}
```

The function *evalFunction* utilizes the *foreach* function in the *foreach package* package to parallelize the execution of *singleEval*. This optimizes the speed of the algorithm.

Next, the criterion values are ranked from lowest to highest for each individual/model. The probabilities are determined directly from the rank:

$$Pi = \frac{-rank_i}{\sum_{i=1}^{n} N}$$

Where N is the population size (or the number of individuals in the model).

The function returns a matrix that contains the fitness level of each individual/model, the rank for each model, and the sampling probability for each model.

updateSamp function

The *updateSamp* function selects n/2 pairs from the parents individuals/models from the previous generation according to the fitness level. The sampling probabilities are determined by output from the *evalFunction* from the previous iteration of the algorithm. Fitter individuals/models have a greater chance of being selected.

crossover function

This function performs crossover for one pair of individuals/models depending on a crossover probability. It randomly generates a cutoff value and then crossover is performed. See the following example:

$$Individual_1 = 1\ 0\ 1\ 1\ 0\ 1$$

$$Individual_2 = 0\ 1\ 1\ 0\ 0\ 1$$

Random Cutoff = 2, now we have:

$$Individual_1 = 1\ 0\ 1\ 0\ 0\ 1$$

$$Individual_2 = 0\ 1\ 1\ 1\ 0\ 1$$

mutation function

This function performs mutation on a pair of individuals/models depending on a mutation probability. A mutation is a switch from a 1 to a 0 or a 0 to a 1 in an individual/model (in other words, it's the act of changing the inclusion or exclusion of a variable in a given model).

Mutation can only occur in positions where both parents shared the same value. The following is an example of boxes are the positions in which mutation could occur:

$$Individual_1 = 1\ 0\ \boxed{1}\ 0\ \boxed{0}\ \boxed{1}$$

$$Individual_2 = 0\ 1\ \boxed{1}\ 1\ \boxed{0}\ \boxed{1}$$

Now for each of the following positions, mutations will occur at the user specified $mRate$. For example, if $mRate = .01$ and the last position in $Individual_2$ is selected to be mutated, we will have the following output:

$$Individual_1 = 1\ 0\ 1\ 0\ 0\ 1$$

$$Individual_2 = 0\ 1\ 1\ 1\ 0\ \boxed{0}$$

select function

This is the main function that implements the genetic algorithm. The functions are implemented in the following order:

1. *popInit* to initialize the population
2. *evalFunction* to get the initial sampling probabilities for each model
3. Repeat until convergence, or the maximum number of iterations have been completed:
   (a) *updateSamp* sample from the
   (b) *crossover* execute in a loop to iterate over all of the pairs in the population
   (c) *mutation* execute in a loop to iterate over all of the pairs in the population
   (d) *evalFunction* execute to get the sampling probabilities for each individual/model

3 Testing

4 Contributions