Writeup

Duncan Black

12/16/2021

Abstract

In this project we designed and tested a genetic algorithm for selecting relevant regression variables. The algorithm is called genetic because it uses biological principles such as selection, crossover, and mutation in order to converge to a better fitted model, with less unnecessary variables, each new generation. Our work consists of a main selection function and six helper functions in which it relies on. Finally, we wrote an auxiliary data drawing function allowing us to easily draw testing data. The technical elements of the algorithm design were drawn from the *Givens and Hoeting* text provided.

Developmental Approach

The first step in ensuring our package was well-formulated was to settle the discussion of whether to use a functional or object-oriented approach. While there were certainly aspects of the project applicable to an OOP design, the three of us immediately gravitated towards a functional system, partly as a consequence of ease with which it allowed us to translate the textual documentation of the genetic generation algorithm. For data structures, the primary input data was a simple dataframe, while we chose to represent the different combinations of component vectors to be used in the linear model as strings of binary integers. Thus, the population of each generation consisted of a list of strings of 1's and 0's. In our implementation, the different stages of the generation process – selection, crossover, and mutation, each took the form of a different function operating on these strings. In this way, operations on the population were not only efficient in that they could be vectorized, but also because their inputs and outputs were limited to spatially efficient character strings. Furthermore, this modular approach allows users to input their own forms of each step, in keeping with the text's suggestion that genetic algorithm research is under continual development.

The package is built using the standard functionality provided by RStudio - this helps with creating the necessary components of our package (R, man, NAMESPACE, DESCRIPTION, etc.). We split our code into two different .R files in the R folder - the main function (select) and the supporting functions we used. In order to import the package dependencies (dplyr, stringr) we need, we import them in the NAMESPACE file and state them in the DESCRIPTION file. We export only the `select` function, as this is the main function for which we wrote many supporting functions. This is the only function of the package we allow the user to interface with. We use roxygen2 to generate the .Rd file for our select function, using inline code comments to fill out the fields of the documentation. We have our formal testing of the package in the tests folder. We build the package by using the devtools::build() command, which generates our .tar.gz file.

In order to install this package, please run:

```
devtools::install_github('duncan-black/GA', host = "github.berkeley.edu/api/v3",
                         auth_token = your_auth_token_string_here)
```

Generate an auth_token for access to the private repository on GitHub prior to running this command.

The primary function, `select`, found in `/R/select.R`, requires three input variable: `x`, a dataframe, `reg`, a regression type, and `reg_fun`, the formula to be used for mapping the data to the given regression. The function also allows users to input pointers to different supporting functions to be used in stead of those written in the package, among other optional specifications. `select` then type-checks these inputs for consistency. It then randomly generates a starting population of binary strings, and then iteratively calls the `generation` function on the population until either convergence or an arbitrary maximum number of generations is reached. It returns both the binary string representing the optimal combination found, as well as the dataset restricted to those components.

The `generation` function is simply a wrapper that calls the functions for the three steps of the genetic algorithm. If in the final generation, it skips crossover and mutation in order to lower diversity and maximize the probability of accurate convergence.

The selection function, `gen_selection` is the most complex in the package. Using `lapply` it first creates a copy of the data set with the columns denoted by 0 in the binary string removed, and calls a wrapper function, `fitness`, that returns the fitness of a binary combination of components with respect to the (column restricted) data and model. It then, per the textual reference, assigns a probability to each combination based on that combination's fitness value's rank within the greater population. Finally, it builds a random sample of binary combinations based on this probability distribution and sorts it by fitness.

The crossover function, `gen_crossover`, pairs adjacent binary strings, assigns each pair a random index, and splices each at this index. It then swaps their starts and finishes. While this function is implemented iteratively rather than through a vectorized approach, the relatively small scale of the function limits the cost to the overall efficiency.

The mutation function, `gen_mutate`, is the simplest function. It probabilistically toggles one in every C characters in the population (1 turned to 0 and 0 to 1), where C is the number of characters in each binary string.

The function for drawing simulated data `testdata` is a moderately complicated but self contained auxiliary function. Its inputs allow the user to tune many different possible scenarios. These include the total number of variables, the percentage of variables that are relevant to the response variable, the number of data points, the amount of noise, the link function, the drawing distribution, and the range of covariate values. In general it works by first drawing the relevant set of covariates and using them to make a response variable with added gaussian noise. It then draws some irrelevant covariates which it attaches to the dataset and shuffles. It returns a test dataset along with the "true" binary string, and variable parameters.

One of our main goals was flexibility and modularity allowing users to input both their own regression function and their own fitness function. This is along with the ability to input your own selection, crossover, and mutation function. In order to have some structure we assumed these custom functions would have the same input and outputs as the standard ones. This restriction allowed us to keep our code concise and reliable. Our error checking for the main function just confirms that all the inputs are of the proper type and form giving helpful error messages if they are not.

This was made easier by the way we segmented our main and helper functions. The selection, crossover, mutation, and fitness functions are self contained helper functions. This allows easy modification and gives our main function general support for any modified versions of these functions.

Testing/Resulting Functionality

In order to formally test our functions, we use the tests file of the package. We have a series of tests for the functions we wrote for the implementation of the genetic algorithm. We check our functions for some kind of expected output, given some test data. For our main select function, we test with data that we

generate (of which we know the true binary string) and we test to see if our genetic algorithm will converge to the true value. For this, we tested with different fitness functions, such as AIC and BIC. At one point, we tried testing with `lm` instead of `glm`, but it generated over 10000 warnings (many for each iteration of the algorithm), and even though it still converged, we chose not to include it in our tests. We also checked that it will give useful errors when the inputs are not valid for an invocation of the function. For each of the helper functions, we test their functionality by checking the type and dimensionality of the output.

In order to check for the correct inputs in each function, we check for the expected type of the passed parameters in the function itself.

In order to run all the tests of the package, please try this invocation: `testthat::test_package("GA")`. If that doesn't work, please try to use test_dir to test the functions straight from the file/directory:

'https://testthat.r-lib.org/reference/test_dir.html'

The tests are located in the 'GA/tests/testthat/' directory. Overall, please expect the tests to take 2-3 minutes.

Note: We have a strange issue with the select function; randomly, in any iteration of the `gen_select` function, there is a (low) chance of an error and a warning that has to do with our `col_filter` function introducing NAs. We have tried to find any cause to this error to no avail. In particular, even if a test has failed to due this error, it will pass if simply run again identically.

We worked with N the number of generations and P the population in order to find a balance between efficiency and accuracy. We settled on a default generation number of forty and a population around eight times the generations that would reduce with successive generations. Population in particular affected the time and accuracy of the algorithm a lot. A population of around eight times the generation number left a large enough final population to give reasonable accuracy. This allowed correct convergence for small numbers of variables, around six, without having to have a large number of generations. For larger data, around 20 variables, the algorithm's default number of generations gives reasonable but imperfect accuracy.

In the code below you can see the results of our algorithm on two examples:

```
library(GA)

#testdata function (also found in the supporting functions GA package)
testdata = function(nvar,rper,npoints,noisevar,linkfunc,dist,minx,maxx){

  # True Response/Related Predictors
  beta = sample(1:5,nvar*rper,replace=TRUE)
  beta0 = sample(1:5,1,replace=TRUE)
  X_selected = matrix(runif(npoints*(nvar*rper),min=minx,max=maxx),ncol=(nvar*rper))

  # Linear Response
  Y_nonoise = X_selected%*%beta + beta0

  # Unrelated Predictors
  if(rper != 1) {
    ncol = ceiling(nvar*(1-rper))
    X = cbind(X_selected,matrix(runif(npoints*ncol,min=minx,max=maxx),ncol=ncol))
  }
  else {
    X = X_selected
  }
```

```
  ##Types of Regressions
  Y = Y_nonoise + rnorm(length(Y_nonoise),0,noisevar)
  Y = linkfunc(Y)

  if(dist == 'Poisson') {
    #Note no error term due to poisson variance
    print("With poisson regression noisevar will be ignored")
   for(i in 1:length(Y)) {
      Y[i] = rpois(1,Y_nonoise[i])
    }
  }
  else if(dist == 'Bernoulli') {
    for(i in 1:length(Y)) {
      Y[i] = rbinom(1,1,Y[i])
    }
  }
  else if(dist == 'Gaussian') {
    Y = Y
  }
  else {
    stop("Error: Invalid distribution")
  }

  #Collect into Dataframe
  Y = as.data.frame(Y)
  X = as.data.frame(X)

  #Scramble relavent variable placements
  indexscramble = sample(1:nvar,nvar,replace=FALSE)
  X = X[indexscramble]
  bin = c(as.character(rep(1,nvar*rper)),as.character(rep(0,(nvar-nvar*rper))))
  bin = bin[indexscramble]
  colnames(X) = paste0("X",as.character(c(1:nvar)))
  colnames(Y) = c("Y")

  #Return "true" variables and data
  bin = paste(bin, collapse = '')
  data = cbind(X,Y)
  beta = c(beta[indexscramble])

  return(list('data'=data,'bin'=bin,'beta'= beta,'beta0' = beta0))
}

x_small <- testdata(nvar = 6, rper = 0.5, npoints = 100, noisevar = 1, linkfunc
                    = identity, dist = 'Gaussian', -10, 10)
x_large <- testdata(nvar = 20, rper = 0.5, npoints = 100, noisevar = 1, linkfunc
                    = identity, dist = 'Gaussian', -10, 10)

#True binary
print(x_small[[2]])
```

```
## [1] "100011"
```

```
print(x_large[[2]])
```

```
## [1] "01010110111001000101"
```

```
#Data
x_small <- x_small$data
x_large <- x_large$data

reg <- glm
reg_fun_small <- Y ~ X1 + X2 + X3 + X4 + X5 + X6
reg_fun_large <- Y ~ X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10 + X11 +
  X12 + X13 + X14 + X15 + X16 + X17 + X18 + X19 + X20

family <- "gaussian"
fun <- AIC

out_small <- select(x_small, reg, reg_fun_small, family, fun, ngen=30)
out_large <- select(x_large, reg, reg_fun_large, family, fun, ngen=30)

#Predicted binary
print(out_small[[1]])
```

```
## [1] "100011"
```

```
print(out_large[[1]])
```

```
## [1] "01011111111011100101"
```

Prior testing on the algorithm convergence also brought some unexpected results. The first was that the AIC often differed very slightly from an approximate answer and the exact answer. Also noiseless data seemed to have issues with approximate answers getting better fits than exact answers. For our original smaller population value of four times the generation number the algorithm appeared to struggle with getting a good answer even if we made our noise very small.

Division of Labor

Our work on this project was very collaborative. Each component of building the package was discussed carefully among our group. As such, we conferred with each other very often. At our initial meeting, we assigned each member individual responsibilities on which to direct their main focus:

Hiro was responsible for fleshing out the structure of the genetic algorithm and what the code for it would entail. He initiated the writing of the main functions as well as the key helper functions. He was responsible for making sure that each function would work well with another to support the main 'select' function given a multitude of different parameters.

Duncan was responsible for finding a way to generate data that would test our genetic algorithm implementation. He worked closely with Hiro to debug errors in the code and assessed the ability of our implementation of the genetic algorithm to converge accurately given some known truth in our generated test data. He was also responsible for handling checking for the correct inputs given by the user.

Wei was responsible for creating the structure of the package, including setting up the formal tests, function documentation, and the way the end user would interface with the project. Additionally, he was

responsible for maintaining the package files during the package creation process, making sure the tests and documentation of the package adapted with the changing code.

Although we were assigned these responsibilities, we were able to communicate well with each other on each step to create the end product.