STAT243, Final Project

Angela Song, Khachatur Mirijanyan, Tomas Meade

12/12/2021

Introduction and Installation

Our package is in the GitHub of username angela-song, at the url https://github.berkeley.edu/angela-song/GA. Please install our package as follows, to include our testing scripts with the installation:

```
library(remotes)
install_github(paste0("angela-song", "/GA"),
               host = 'github.berkeley.edu/api/v3',
               auth_token = "f738dd6434f661f20edea2ceb81e22293c5b71fa",
               INSTALL_opts = "--install-tests")
```

We echo the code from the Examples section of the `select()` man page below. Note that `large_testing_data` contains 13 potential predictor variables, where the response variable is the 14th variable and was generated using the `x5`, `x8`, and `x13` variables. Similarly, `huge_testing_data$df` contains 50 potential predictor variables, where the response variable is the 51st variable and was generated using the `X15`, `X26`, and `X45` variables.

```
data(large_testing_data)
select(large_testing_data, 14, n_ind = 4*13)

data(huge_testing_data)
select(huge_testing_data$df, 51, abs_conv_thresh = 0.1)
```

The package may be tested as follows:

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

Approach

We took a functional approach to the code, with no OOP methods. We wrote our functions so that each helper function completes one stage of the genetic algorithm, and our `select` function simply strings calls to the helper functions together. So we ended up with one R script for each of the following components: the main `select` function, first generation initialization, fitness evaluation, parent selection, chromosome crossover, and chromosome mutation. There is also a script that generated our final test/example datasets.

Throughout the algorithm, each generation of "individuals" is represented by a vector of binary strings. This way, unnecessary information from past generations is not stored as future generations are being processed.

In the following, we briefly describe the purpose and design of each individual script:

select.R

The `select` function largely strings together calls to the helper functions to initiate the first generation of individuals, then create new generations until convergence is reached. Outside of calling helper functions, it tracks the minimum objective function value of each generation in order to evaluate convergence. If

the standard deviation of these values among the last several generations is less than a threshold, where the number of generations to check and the threshold are given, then the algorithm is considered to have converged, and the function returns.

init_first_gen.R

To initialize the first generation, our `init_first_gen` function creates a population of specified size of random individuals. Each individual in the population has the specified number of genes and each gene is selected randomly and independently from $\{0, 1\}$.

eval_fit.R

To evaluate the objective function value of each individual in a generation, we created helper function `eval_aic` to compute the AIC of the GLM corresponding to one individual. The main function `eval_fit` then applies `eval_aic` to all individuals in the current generation using `sapply`, if the user does not provide their own objective function. If the user provides an objective function, `eval_fit` applies the function to all individuals using `sapply`.

select_parents.R

The `select_parents` function selects pairs of parents for breeding in each generation. For the selection of parents we used a rank-based fitness function. The fitness function ranks individuals in each generation by their objective function values in descending order. For a generation of size P, the individual with the lowest objective function value would have rank P, the individual with the second lowest objective function value would have rank $P - 1$, and so on, up to the individual with the highest objective function value who would have rank 1. Then the probability of an individual being selected is the rank of the individual divided by the sum of all ranks.

We selected the first parent in each pair according to the probabilities computed above, then selected the second parent uniformly at random from the remaining individuals in the generation. For a generation of size P, the results of selection would be $P/2$ pairs of parents.

We used rank-based selection because, according to the Givens and Hoeting book, rank-based methods are effective in avoiding premature convergence. We selected the second parent uniformly at random to promote genetic diversity.

crossover.R

The default `crossover` functionality uses a random one point split method. A random point between two adjacent loci is selected, and both parents' chromosomes are split at this point. The first segment of the first parent's chromosome is "glued" to the second segment of the second parent's chromosome, and the first segment of the second parent's chromosome is "glued" to the second segment of the first parent's chromosome to yield two child chromosomes. To test functionality, an additional argument can be given to `crossover` that fixes the split points for each parent pair.

The function also allows for a user-defined genetic operator. If the user provides a genetic operator function, the function is applied to pairs of parents using `sapply`.

mutate.R

We originally created a mutate function that mutates every gene of every individual at the specified mutation rate. We liked that this was simple and worked in producing genetic diversity. However, we concluded that this process could be sped up using a slight variation on mutation. To compare, we constructed a different mutate function that selects individuals according to the mutation rate and then changes one randomly selected gene of each selected individual. This method was considerably faster and produced similar results, so we decided to use the faster function in our final package.

The `mutate_all` function implements the faster method described above.

sample_datasets.R

This script generates three datasets with three functions; `get_simple_dataset`, `get_large_dataset`, and `get_huge_dataset`. We used these functions to generate our final test/example data.

The second function is very similar to the first, only larger. `get_large_dataset` creates a data frame consisting of 13 potential predictor variables with 1000 observations and 1 response variable. The potential predictor variables are randomly sampled from hard-coded distributions and hard-coded parameters. The response variable is a fixed linear combination of 3 chosen predictors variables (`x5`, `x8`, and `x13`), plus some random error. The simple dataset is exactly the same as the large one, except it has only 9 potential predictor variables and the response variable is a linear combination of only 2 variables. The functions then return the data frame.

`get_huge_dataset` is much larger and much more random. It consists of 50 potential predictor variables with 1000 observations and 1 predictor variable. Each potential predictor variable is sampled from some unknown random distribution with unknown random parameters. The response variable is still a linear combination of three predictors, but the predictors chosen are also random. The function returns the data frame as well as the column indices of the predictor variables that generated the response variable.

Testing

We tested each helper function called in `select` and `select` itself. The approach for the non-stochastic functions was to simply compare their outputs to known values. For functions with random outputs, we checked against expected values given large random testing sets, or checked against carefully chosen known values. We wrote one test script per function, which we briefly describe below:

test-select.R

The purpose of this test script was to see the accuracy and speed of our genetic algorithm compared to known quantities. This resulted in 2 main tests.

The first test compared the results of using `select` on a randomly generated, fairly large dataset, versus using a naive variable search approach on the same dataset. The dataset came from the `get_large_dataset` function, where the predictor variables that generated the response variable are known (see above for a description of `get_large_dataset`).

With this knowledge, the genetic algorithm was run and both the time taken and the output AIC score were recorded. Then a naive search for the best possible AIC score was also performed, which calculated the 2^{13} AIC scores for all possible variable combinations and found the definitive best AIC score. The time taken was also recorded.

There were 3 expectations within the first test. The first was that the 3 true predictor variables would be included in the output of `select`. The second expectation looked at accuracy. While `select` might not find the best AIC score, it should come fairly close. The third expectation looked at timing, expecting the genetic algorithm to be faster than the naive variable search.

The second test was similar to the first, but it used a different dataset and had slightly different expectations. The dataset came from the `get_huge_dataset` function (see above for a description of `get_huge_dataset`). We compared the output of `select` to the output of `stepAIC` from the `MASS` package.

There were only 2 expectations in the second test. The first was the same as before, and checked that `select` included the true predictor variables in its output. The second expectation was that the AIC score output by `select` was close to what was generated by `stepAIC`. Given the large number of variables, we expected the AIC scores to be close, but not quite as close as in the previous test, so we left a little more room for error.

test-init_first_gen.R

To test `init_first_gen`, we called `init_first_gen` and checked that the size of the resulting generation and the number of genes per individual matched the input arguments.

test-eval_fit.R

To test `eval_fit`, we created small test datasets and checked that `eval_fit` correctly returned the AIC score for different individuals and for different models (we specifically tested LMs and binomial GLMs). We also tested that `eval_fit` correctly returned user-provided objective function values when given a very simple alternate objective function.

test-select_parents.R

To test the parent selection, we created a small test generation of parents and assigned each parent an objective function value. We then calculated by hand what their probability of selection would be according to the rank-based fitness function. We simulated selecting parents using the `select_parents` function a large number of times and verified that the proportion of times each parent was selected was close to their known probability of selection that we calculated analytically.

test-crossover.R

We tested default `crossover` functionality by creating a small test dataset of parent pairs and fixing the split points for each set of parents, thus determining the child chromosomes (see above for a description of simple crossover). Given the split points, we checked that the output of `crossover` (when not given a user-provided genetic operator) matched the known child chromosomes.

We also checked the functionality of `crossover` when given a very simple alternate genetic operator.

test-mutate.R

To test the `mutate_all` function, we created a large test population of one-gene (one-digit) individuals and then applied the `mutate_all` function to it with a specified mutation probability. We then checked that the proportion of individuals who were changed was similar to the input mutation rate.

Results of Applying Our Implementation to Examples

We describe the results of one application of `select` to our examples, where the application was done in the SCF. Due to the random nature of `select`, additional applications will likely yield different numbers. However, the following results are representative of results we were seeing during repeated testing. Please see above for how to run our examples and how the example datasets were generated.

On the large dataset of 13 potential predictor variables (`large_testing_data`), the genetic algorithm took 6.2 seconds and 26 iterations to converge. This resulted in an AIC score of -893.37, which is the lowest possible score. The algorithm also correctly included the 3 true predictor variables in its solution.

On the huge dataset of 50 potential predictor variables (`huge_testing_data`), the genetic algorithm took 71.4 seconds and 79 iterations. This resulted in a score of -975.55, compared to the score generated by `stepAIC` that was also -975.55. In this case the scores were the same between the two methods, but sometimes the AIC score from the genetic algorithm can be a little higher. The algorithm correctly included the 3 true predictor variables in its solution.

The results above demonstrate that `select` can correctly select "true" predictor variables, resulting in AIC scores that are comparable to other, established approaches to variable selection.

Contributions

Our work towards this package can be split into three main categories: coding functionality for our genetic algorithm, testing, and miscellaneous package building and documentation. Our contributions were as follows:

- Function Files:
  - select.R - Angela
  - init_first_gen.R - Angela

  - eval_fit.R - Angela
  - select_parents.R - Tomas
  - crossover.R - Angela, Khachatur
  - mutate.R - Tomas
  - sample_datasets.R - Khachatur
- Test Files:
  - test-select.R - Khachatur, Angela
  - test-init_first_gen.R - Angela
  - test-eval_fit.R - Angela
  - test-select_parents.R - Tomas
  - test-crossover.R - Angela, Khachatur
  - test-mutate.R - Angela, Khachatur
- Miscellaneous:
  - Package Setup - Angela
  - Documentation - Angela
  - Internal Example/Testing Data - Khachatur, Angela
  - Write Up - Angela, Tomas, Khachatur