Stat243 Final Group Project

Cameron Adams, Yuwen Chen, Weijie Xu, Yilin Zhou

December 14, 2017

Please note that the Github username that we are using for this project is "808camBerk".

I. Approach Taken

To perform feature selection in regression problems using genetic algorithm, we conduct the following steps sequentially:

1. Initialization: create the first generation (*generate_founders*).
2. Evaluation: assess the fitness of each chromosome in a generation, and rank the chromosomes according to their fitness (*evaluate_fitness*, *rank_objective_function).
3. Selection, Crossover, and Mutation: mate pairs are selected according to fitness rank, with more fit parents having greater probability of selection and generate a new generation using crossover and possible mutation (*create_next_generation*, *select_parents*, *crossover_parents*, *mutate_child*).
4. Looping and termination: repeat steps 2 and 3 until either the maximum number of iterations is reached or the algorithm converges (*select*).

Step 1: Initialization

We create the first generation with the function *generate_founders*. For binary gene encoding, Givens and Hoeting recommend generation population size P, proportional to C, the number of predictors variables or genes, so that $C < P \leq 2C$. Also, since P is between 10 and 200 in most real applications, we set 10 and 200 as the lower and upper bounds of P. Users can specify their own generation size, but a warning will appear if it is > 200.

To generate founding chromosomes, generation_t0, we randomly over sample $(1.2 \times C \times P)$ 0's and 1's, splitting them up into the P chromosomes to make sure we generate at least P unique values. We then select unique chromosomes and make sure no chromosomes contain all zeros.

Step 2: Evaluation

Fitness evaluation of chromosomes is done by *evaluate_fitness*. Models were evaluated using *lm()* or *glm()* depending on users input for the argument *family*, the default family is Gaussian. Default fitness evaluation is completed using *sapply()* and the *objective_function* which takes as input a function which evaluates an fitted model object for an objective function. The default objective_function is *stats::AIC*. If users specify *nCores* > 1, they can parallelize the lm/glm computation across a specified number of cores. Parallization is done using *mclapply* using prescheduling.

After we have computed the objective function for all chromosomes in a generation, we rank them using *rank_objective_function*. This function ranks chromosomes objective function score, (default is AIC), by whether the algorithm is optimizing towards a minimum (default) or maximum. This function returns the chromosome number, rank from 1:P (P is "best"), and objective function output.

Step 3: Selection, Crossover, and Mutation

In the third step of our algorithm, we create a new generation by breeding parents of the previous generation using genetic concepts of selection, crossover and mutation with the function *create_next_generation*. The population size of a each generation is fixed and will be same across all generations. A while loop is used to generate children using selection, crossover, and mutation process. As with parents, no children will have all zero chromosomes.

Selection. Parent mate pairs are selected randomly with replacement according to their fitness (*select_parents*). Each parent has a probability of selection proportional to their fitness rank:

$$\phi_i = \frac{2 * r_i}{P * (P + 1)},$$

where, P is the size of current generation, and r_i is a vector ranks for each parent chromosome. This ranking method gives the median candidate a selection probability of approximately 1/P, and the best candidate has probability of selection equal to 2/(P+1). This ensures that parents with better fitness are more likely to be selected and that selection will not be over deterministic. Selecting only parents with high fitness could lead to a bottleneck population and produce generations that can only converge to local optima. The first parent is selected at random using rank probabilities ϕ_i. The second parent is selected at random with all parents having uniform probability of selection. A chromosome cannot pair with it self.

Crossover. We implement three methods for crossover to generate two children from each mate pair (*crossover_parents*). Users can specify a probability of crossover occurring between a mate pair, *pCrossover* (default pCrossover = 0.8). *Rbinom()* is used to determine if crossover will occur (1 = yes, 0 = no), using this probability. If no, parents pass their chromosomes on directly to the next generation. If yes, crossover occurs.

1. First method. In this method, we used multipoint crossover to generate two children. We randomly sample three non-overlapping points within the length of the chromosome for crossover. Parents swap genes at each segment, with each segment alternating direction of swap, rendering two children. This method is a classical implementation of genetic crossover.

2. Second method. In this method, we assign probabilities for each parent's genes according to their fitness rank. The parent with the with higher fitness have a higher probability or passing on their genes to children:

$$C = P_1 \times \frac{P_1^r}{P_1^r + P_2^r} + P_2 \times \frac{P_2^r}{P_1^r + P_2^r},$$

where, P_1 and P_2 are the parent chromosomes, and P_1^r and P_2^r are the parent ranks, and C rethe probabilities for each parent gene for a child. We use *rbinom()* to sample genes for each child according these probabilities.

3. Third Method. In this method, non-concordant genes between parents, reprobablistically weighted according to parent ranks. The first child has probabilities proportional to the first parent rank, the second child according the second parent rank.

```
    else if (crossover_method == "method3") {
        #METHOD 3 ----------------
        child1 <- parent1
        child2 <- parent2
        child1[parent1 != parent2] <-
            stats::rbinom(sum(parent1 - parent2 != 0), 1,
                   prob = parent1r / (parent1r + parent2r))
        child2[parent1 != parent2] <-
            stats::rbinom(sum(parent1 - parent2 != 0), 1,
                   prob = parent2r / (parent1r + parent2r))
    }
```

Mutation. After selection and crossover, we perform a mutation step (*mutate_child*) whereby polymorphisms are randomly inserted into child chromosomes according to specified probability of mutation. The default mutation rate is proportional to the size of population, P, and length of chromosome, C:

$$M = \frac{1}{P * \sqrt{C}},$$

where, M is the probability of crossover. Mutation gene targets are randomly generated using *Rbinom*() with the probability of mutation for each gene position. To perform mutation, we then take the absolute value of the difference between a child chromosome and the mutation gene targets. User's can specify their own mutation rate, however high mutation rates will prevent the algorithm from converging and producing an answer.

Following selection, crossover and mutation, we have *generation_t1*.

Step 4: Looping and Termination

After creating and evaluation initial generation, a while loop is used to create and evaluate successive generations. Users can specify if they want to test for convergence or let the algorithm run for the specified number of iterations (default converge = TRUE). The algorithm will converge if the objective function score for the highest ranked individual in generation $t + 1$ is equal to the mean of the scores of the top 10% of chromosomes in the $t + 1$ and t generations (within default tolerance = 5e-4). This converge criteria indicates that the algorithm is no longer producing new optimal solutions, and there is little variability in each the recent generations.

The function *select* is the main function for our package. Users specify the predictor variables (X) for variable selection and a response variable (Y). Other function arguments allow users to control how the algorithm operates. Users can specify the type of generalized linear model using *family*, crossover method, and functions for objective functions and crossover can be specified by the user if they do not want to used the default implementation. They can also specify parallelization (nCores > 1), probability of crossover (pCrossover), population size (start_chrom), mutation rate (mutation_rate), optimize to convergence (converge), conference criteria tolerance (tol), to minimize of maximize objective function (minimize), and whether to print function output (verbose).

This function has the following steps:

1. Error check argument inputs
2. Generate founding population (*generate_founders*)
3. Evaluate initial population (*evaluate_fitness*, *rank_objective_function*)
4. Loop to generate successive generations until convergence or maximum iterations are complete. (*evaluate_fitness*, *rank_objective_function*, *create_next_generation*, *select_parents*, *crossover_parents*, *mutate_child*)
5. Return output with optimum and other data concerning algorithm performance.

This function outputs an list of class "GA", which contains:

- *Best_model*: the optimum model
- *optimize*: information about the objective function, the optimum, minimize or maximize, and crossover method
- *iter*, number of iterations
- *converged*, if converged
- *converge_data*, objective function scores, and rankings for each generation
- *timing* system run time for each generation.

We wrote an S3 plot method for output of selection for the class "GA" (*plot.GA*). It takes in a *GA* object and produces a plot showing objective function scores for each chromosome for all generations, and mean and "best" objective function scores across all generations.

II. Evaluation

Simulated data. We simulated data using *simrel* package. The package allows us to generate data sets and identify the total number of predictors and number of relevant predictors and components. This makes it easy to test "truth" (actual relevant predictors) against what our algorithm produces.

```
########
# Evaluation

library(simrel, quietly = TRUE)
##
## Attaching package: 'DoE.base'
## The following objects are masked from 'package:stats':
##
##     aov, lm
## The following object is masked from 'package:graphics':
##
##     plot.design
## The following object is masked from 'package:base':
##
##     lengths
##
## Attaching package: 'sfsmisc'
## The following objects are masked from 'package:conf.design':
##
##     factorize, primes
library(GA, quietly = TRUE)
##
## Attaching package: 'GA'
## The following object is masked _by_ '.GlobalEnv':
##
##     generate_founders

# Simulate large number of predictors ----------------
set.seed(84)
n <- 500 # number obs
p <- 50 # number predictors
m <- 25 # number relevant latent components
q <- 25 # number relevant predictors
gamma <- 0.2 # speed of decline in eigenvalues
R2 <- 0.75 # theoretical R-squared according to the true linear model
relpos <- sample(1:p, m, replace = F) # positions of m
dat <- simrel(n, p, m, q, relpos, gamma, R2) # generate data
x <- dat$X
y <- dat$Y

# run GA::select
meth1 <- GA::select(y, x, crossover_method = "method1", verbose = FALSE)
meth2 <- GA::select(y, x, crossover_method = "method2", verbose = FALSE)
meth3 <- GA::select(y, x, crossover_method = "method3", verbose = FALSE)

# simulated relevant predictors
dat$relpred
##  [1]  2  3  4  6  7  8 10 11 13 14 15 17 19 26 30 32 34 35 37 38 39 42 45
```

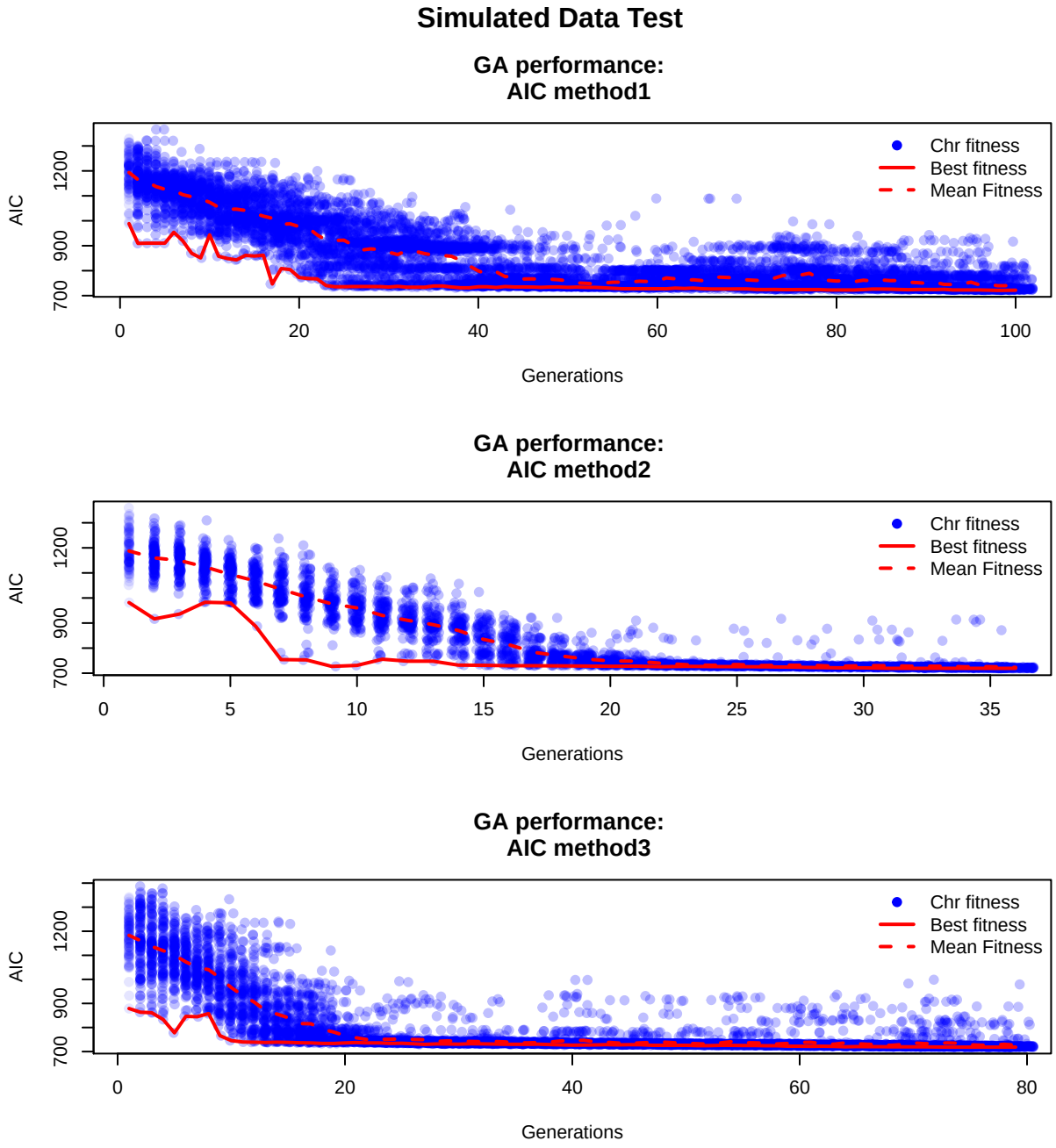
```
## [24] 48 50

# check outputs
meth1$Best_model
##  [1] "2"  "3"  "4"  "6"  "7"  "8"  "10" "11" "13" "14" "15" "17" "18" "19"
## [15] "20" "26" "30" "32" "34" "35" "36" "37" "38" "39" "41" "42" "43" "45"
## [29] "46" "47" "48" "49" "50"
meth2$Best_model
##  [1] "2"  "3"  "4"  "6"  "7"  "8"  "10" "11" "13" "14" "15" "17" "18" "19"
## [15] "20" "26" "30" "32" "34" "35" "36" "37" "38" "39" "41" "42" "43" "45"
## [29] "48" "50"
meth3$Best_model
##  [1] "1"  "3"  "4"  "5"  "6"  "7"  "8"  "10" "11" "13" "14" "15" "17" "18"
## [15] "19" "26" "30" "34" "35" "37" "38" "39" "42" "45" "48" "50"

meth1$optimize$value
## [1] 722.8085
meth2$optimize$value
## [1] 717.5552
meth3$optimize$value
## [1] 718.4392

dat$relpred %in% meth1$Best_model; sum(dat$relpred %in% meth1$Best_model)
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [1] 25
dat$relpred %in% meth2$Best_model; sum(dat$relpred %in% meth2$Best_model)
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [1] 25
dat$relpred %in% meth3$Best_model; sum(dat$relpred %in% meth3$Best_model)
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE
## [1] 23

par(mfrow = c(3, 1), oma = c(0, 0, 2, 0))
plot(meth1)
plot(meth2)
plot(meth3)
mtext(text = "Simulated Data Test", side = 3, outer = T, font = 2)
```



The three crossover methods work well to identify simulated relevant predictors, identifying 25, 25, and 23 of 25 using crossover method1, method2, and method3, respectively. Crossover method2 converged the fastest, which is expected as it generates less diverse generations than method1 and method2.

Real Data Test: Ionosphere. Our first real data test was done using Ionosphere data in the *mlbench* package. We used all observations (n=351) and 32 variables in Ionosphere data set to test performance of our algorithm against *stepAIC()*, a variable selection algorithm that also uses AIC as the objective function.

```
#######
# Real data test #1: Ionosphere

library(mlbench, quietly = TRUE)
library(MASS, quietly = TRUE)
```

```
##
## Attaching package: 'MASS'
## The following object is masked from 'package:GA':
##
##     select
```

```
# get "real" test data
data("Ionosphere")
Ionosphere <- Ionosphere[,-c(1, 2, 35)]
names(Ionosphere)

y <- Ionosphere[, 32] # response
x <- as.matrix(Ionosphere[,-c(32)]) #predictor set

fit <- lm(V34 ~ ., data = Ionosphere)
step <- stepAIC(fit, direction="both", trace = -100)
```

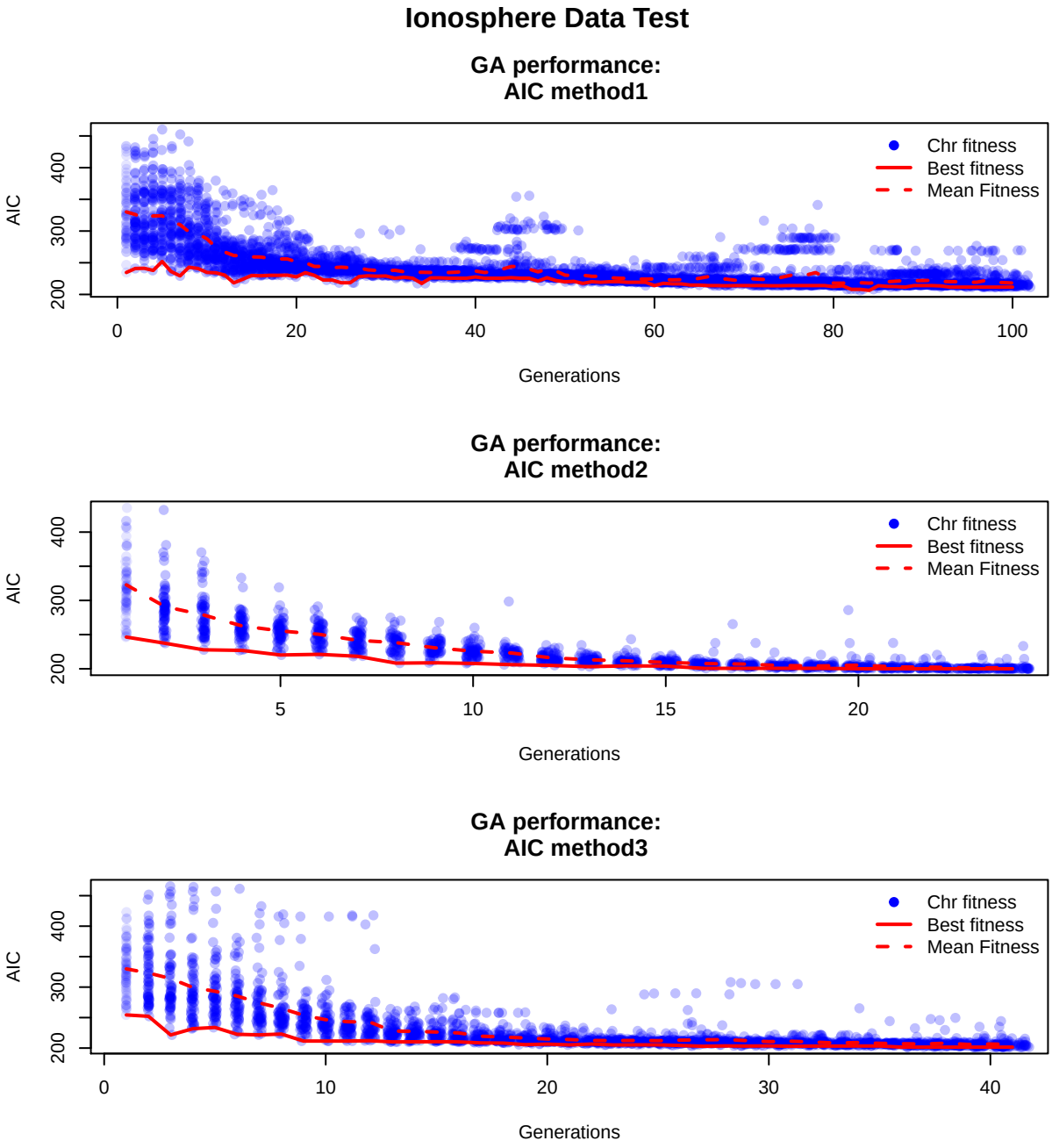
```
#######
# Real data test #2: mtcars

# run GA::select on Ionosphere data
meth1 <- GA:: select(y, x, crossover_method = 'method1', verbose = F)
meth2 <- GA:: select(y, x, crossover_method = 'method2', verbose = F)
meth3 <- GA:: select(y, x, crossover_method = 'method3', verbose = F)

# check AIC's
AIC(step)
## [1] 199.5313
meth1$optimize$value
## [1] 210.3404
meth2$optimize$value
## [1] 199.999
meth3$optimize$value
## [1] 201.5235

# check variable selection
step_preds <- names(step$coefficients)[-1]
step_preds; length(step_preds)
##  [1] "V4"  "V5"  "V7"  "V8"  "V9"  "V11" "V12" "V13" "V14" "V15" "V16"
## [12] "V18" "V19" "V20" "V23" "V25" "V27" "V29" "V30" "V31" "V32" "V33"
## [1] 22
sum(meth1$Best_model %in% step_preds)
## [1] 17
sum(meth2$Best_model %in% step_preds)
## [1] 22
sum(meth3$Best_model %in% step_preds)
## [1] 22

# plots
par(mfrow = c(3, 1), oma = c(0, 0, 2, 0))
plot(meth1)
plot(meth2)
plot(meth3)
mtext(text = "Ionosphere Data Test", side = 3, outer = T, font = 2)
```



Using "real world" Ionosphere data set available through *mlbench*, we tested the performance of our crossover methods against step-wise AIC selection. Our methods performed similarly selecting essentially the same variable and *stepAIC()*. The optimal AIC's found by GA::select were also very similar to what was found by *stepAIC()*.

Evaluation Summary. Overall, our algorithm adequately performed variable selection. Using simulated data, we were able to find virtually all of the "true" predictor variables. The algorithm also performed well on real world data and closely replicated the performance of *stepAIC()*.

III. Members' Contributions

In this project, all members in the group helped test the practicability of each function and propose constructive suggestions in designing the algorithm, especially on the set-up of the crossover and mutation methods, as well as the criteria of convergence. Then each group member is also responsible for the following major parts:

Cameron Adams: wrote and continuously improved the functionality of functions including “select”, developed the first and third crossover methods, wrote the help information and the complted formal testing for functions and package creation.

Yuwen Chen: wrote the formal testing for functions including “generate_founders” and “select”, constructed the GA package and ensured that it is operational, debugged some impracticabilities in the algorithm.

Weijie Xu: constructed the basic structure of the algorithm and wrote functions including “generate_founders”, developed the second crossover method , wrote the formal testing for functions such as “create_next_generation”.

Yilin Zhou: checked the functionality of each step and improved the consistency of the presentation, constructed and wrote this documentation, double checked to ensure the project attains all the requirements.

IV. References

Geof H. Givens, Jennifer A. Hoeting (2013) Combinatorial Optimization (italicize). Chapter 3 of Computational Statistics (italicize).

Henderson and Velleman (1981), Building multiple regression models interactively. Biometrics, 37, 391-411.

Sigillito, V. G., Wing, S. P., Hutton, L. V., Baker, K. B. (1989). Classification of radar returns from the ionosphere using neural networks. Johns Hopkins APL Technical Digest, 10, 262-266