project_report

Implementation

Code of the package resides in https://github.berkeley.edu/kunlint/GA. The main function **select** chooses predictors based on their fitness scores, which are defined to be the inverted values of the fitness function (e.g. AIC/BIC) applied on the fitted lm/glm model using a set of predictors. Each set of predictors is called a subject, and we encode them in a binary fashion: 0 means exclusion of the predictor, and 1 means inclusion of the predictor. The length of the binary vector is the number of predictors. As a starting point, the main function calls the **init** helper function to generate a random starting generation, which consists of binary vectors. The number of binary vectors in a generation is determined by the number of predictors (this can be adjusted using the argument *gen_size* of the main function). After initialization, we repeatedly update the generation until some fixed number of iterations or some stopping criteria. To update the generation, we first use the **select_parents** helper function to choose the parents according to their fitness scores. The scores are transformed to ranks and we assign probabilities of being chosen according to the rank. Inside the **select_parents** function, we call another helper function **fitness_func**, which basically computes the fitness scores of the individual binary factor (The **fitness_func** also calls the **construct_formula** helper function to form a formula to feed to the glm/lm function) . After selection, we perform the cross over using the **cross_over** helper function that exchanges segments of two binary vectors and produces two offspring. The offspring are grouped into a new generation, and we apply the **mutate** function to change the encoding of the vectors from 0 to 1 or from 1 to 0 with a small probability. All of the above produce a new generation of binary vectors.

In the selection stage, there are two modes, the first mode selects both parents according to the ranking of their fitness scores, and the second mode selects the first parent according to the ranking and the other completely random. It is also possible to partially update the generation, a sort-of "elitist" approach, i.e., good-performing predictors are copied exactly from the current generation to the next generation (The ratio to keep is called *generation_gap*). In order to stop the iteration when there is no improvement expected, we define the diversity factor as the ratio of distinct binary vectors we must have in each generation. If the ratio is lower than the diversity factor, and the previous best-performing predictors' fitness score and the current best-performing predictors' fitness score is close enough, we stop the iteration.

It is also possible to specify more complicated cross-over by cutting the two binary vectors in more than 1 cutpoints; this can be done by setting the *cross_point* argument.

In order to save some time computing the fitness scores, we create an environment object to store the fitness scores of distinct binary vectors. This basically acts as a hash table for quick look up. We see a lot of binary vectors appear multiple times, and we don't need to fit and compute the fitness scores again.

Testing

Function "select":

For testing the main select function, we created 2 datasets - the baseball dataset that is found using real-life data and the dummy dataset that is data that is randomly generated. We created 27 test cases that test that every input for the select function runs. For tests that check the behavior of certain inputs like cross_point and generation_gap, we wanted to ensure that changes to those inputs should generally

improve the performance of our algorithm, so we checked that setting or changing those inputs lead to the model correctly identifying the predictors for the dummy dataset. Since our select function involves random processes in the mutate, crossover, and select_parents functions, we set the random seed to 0 to ensure reproducibility for all our tests.

Function "fitness_func":

For testing the fitness_func helper function, we ran it on its default parameters, on various inputted chromosomes, and on various fitness functions that take in a fitted model. For testing that the function runs on other inputted fitness functions used to evaluate the quality of a fitted model on a dataset, we created custom functions that served to score an inputted model in various ways, like a custom SSE function that takes in a fitted model, to ensure that our genetic algorithm is able to take custom-made fitness functions as input, as long as those functions were able to take in a fitted model. Lastly, we ran the fitness_func on two custom fitness functions that do not take in a model as input, to ensure that the helper function would fail to run in those instances, as expected.

Function "construct_formula":

For testing the construct_formula helper function, we ran it on the baseball dataset and a custom dataframe to see if it generates the correct formula to be fed into the fitness function.

Function "select_parents":

I tried different input parameters to see whether the number of columns and rows are the same as the initialized matrix. I set selection_mode to be 1 and 2, as well as different response inputs. Also I intentionally made some wrong inputs to check if there is a warning . Two datasets are used in the test. The first one is the baseball datasets found on the internet, and the second one is a dummy dataset created. I want to see if the function works properly under different datasets. It turns out that the function works very well.

Function "remove_all_zeros":

Three vectors were created to test if the function works when there exists zero, one and two all zero columns.

Function "init":

I test whether the number of rows and columns are what should be expected. An incorrect input is intentionally created to test if there is a wanring.

Function "mutate":

When p is 0, which means that the probability of mutation is 0, the output vector should be the same with the input. When p is 1, which means the probability of mutation is 100%, the output vector should be 100% different from the input. Also, I made a vector with 100000 elements of one and applied the "mutate" function with p=1. It works well because there should be around equal numbers of 0 and 1 in the output vector.

Results

Result of running the algorithm on dummy_data:

In this dataset, there are 10 columns, which means that there are 10 predictors. 1000 rows are included. Response "y" is a linear function of the first five predictors x1 to x5, with an intercept and random noise added. The selection function returns a result after 13 iterations. In the output, the *best_so_far* are the best predictors among all the iterations. The *best_condidate* is the best predictors in the last generation. The function works perfectly because the *best_candidate* in the output are x1 to x5. It tells us that the first five predictors produce the highest fitness score, which is true since y is a linear function of these five predictors.

Here is a sample of that dataset:

```
library(devtools)
```

```
## Loading required package: usethis
```

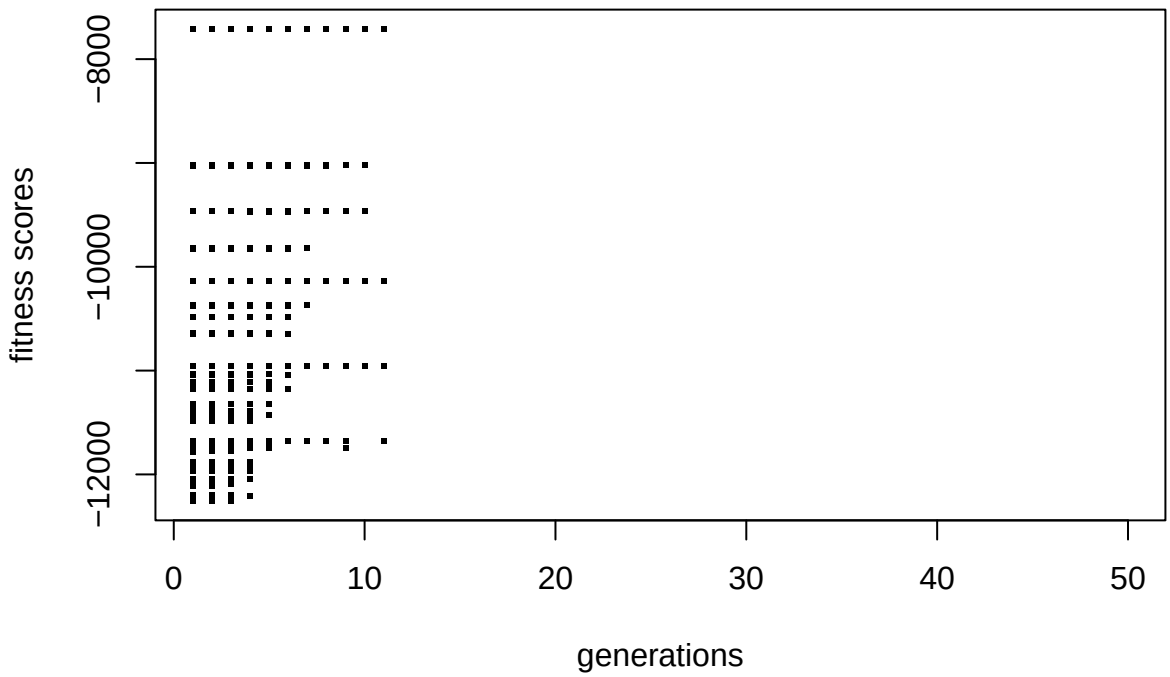
```
load_all()
```

```
## i Loading GA
```

```
str(dummy_data)
```

```
## 'data.frame':    1000 obs. of  11 variables:
##  $ x1 : num  26.9 7.97 11.16 17.19 27.25 ...
##  $ x2 : num  7.97 15.92 20.55 11.5 28.65 ...
##  $ x3 : num  11.6 26.2 29 26 13.1 ...
##  $ x4 : num  15.94 24.07 14.38 5.32 11.91 ...
##  $ x5 : num  20.148 5.633 15.143 0.819 14.889 ...
##  $ x6 : num  8.92 1.06 10.67 7.47 26.38 ...
##  $ x7 : num  8.86 23.1 20.71 19.51 2.24 ...
##  $ x8 : num  10.6 21 13.6 19.4 18.4 ...
##  $ x9 : num  23.58 3.85 3.83 23.33 12.6 ...
##  $ x10: num  12 28.21 29.9 6.99 4.26 ...
##  $ y  : num [1:1000, 1] 279 216 226 117 100 ...
```

```
set.seed(0)
ex <- select(dummy_data,'y',cross_point=3,generation_gap = 0.5,gen_size = 40,
             max_iter = 50,selection_mode=2,details = T)
```



```
print(ex$best_cand)
```

```
## [1] "x1" "x2" "x3" "x4" "x5"
```

```
print(ex$fitness_score)
```

```
## [1] -7705.271
```

```
print(ex$iteration)
```

```
## [1] 11
```

Result of running the algorithm on the baseball dataset:

This dataset contains the statistics such as batting average and number of home runs for 337 players in the 1991 season, along with their salary in 1992. This is the same dataset that was used in Computational Statistics.

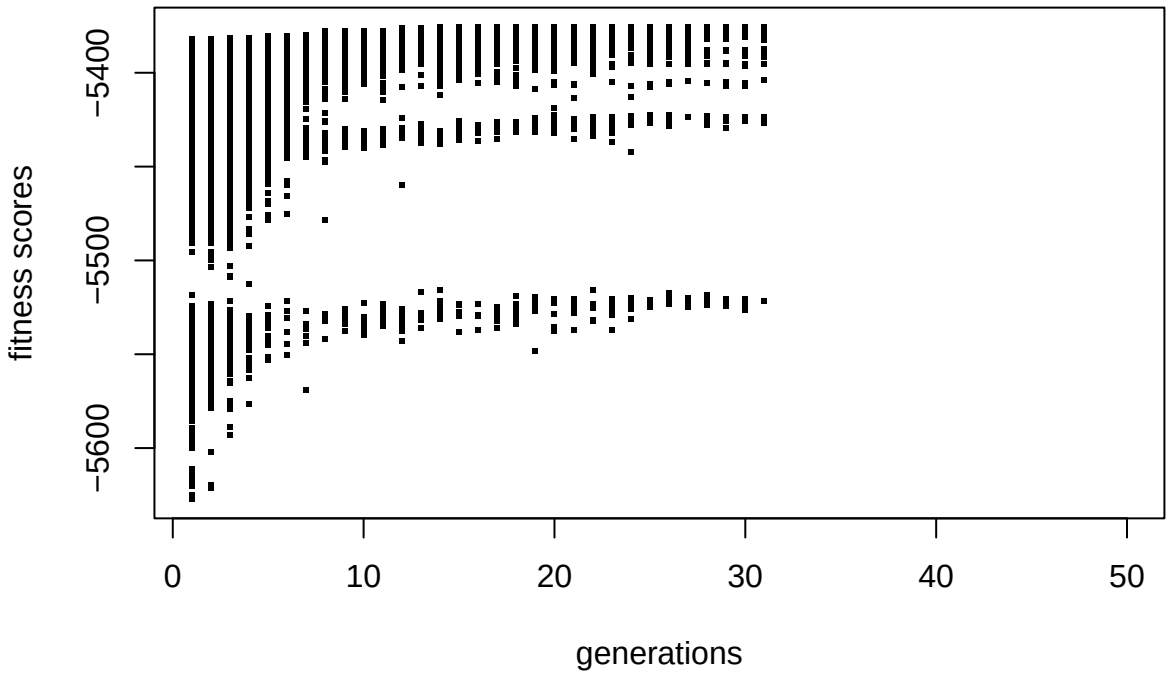
Here is a sample of that dataset:

```
str(baseball)
```

```
## 'data.frame':    337 obs. of  28 variables:
##  $ salary      : int  3300 2600 2500 2475 2313 2175 600 460 240 200 ...
##  $ average     : num  0.272 0.269 0.249 0.26 0.273 0.291 0.258 0.228 0.25 0.203 ...
##  $ obp         : num  0.302 0.335 0.337 0.292 0.346 0.379 0.37 0.279 0.327 0.24 ...
##  $ runs        : int  69 58 54 59 87 104 34 16 40 39 ...
##  $ hits        : int  153 111 115 128 169 170 86 38 61 64 ...
##  $ doubles     : int  21 17 15 22 28 32 14 7 11 10 ...
##  $ triples     : int  4 2 1 7 5 2 1 2 0 1 ...
##  $ homeruns    : int  31 18 17 12 8 26 14 3 1 10 ...
##  $ rbis        : int  104 66 73 50 58 100 38 21 18 33 ...
##  $ walks       : int  22 39 63 23 70 87 15 11 24 14 ...
##  $ sos         : int  80 69 116 64 53 89 45 32 26 96 ...
##  $ sbs         : int  4 0 6 21 3 22 0 2 14 13 ...
##  $ errors      : int  4 4 6 22 9 5 11 4 3 7 ...
##  $ freeagent   : int  1 1 1 0 0 1 1 0 0 0 ...
##  $ arbitration : int  0 0 0 1 1 0 0 0 0 0 ...
##  $ runsperso   : num  0.863 0.841 0.466 0.922 1.641 ...
##  $ hitsperso   : num  1.913 1.609 0.991 2 3.189 ...
##  $ hrsperso    : num  0.388 0.261 0.147 0.188 0.151 ...
##  $ rbisperso   : num  1.3 0.957 0.629 0.781 1.094 ...
##  $ walksperso  : num  0.275 0.565 0.543 0.359 1.321 ...
##  $ obppererror : num  0.0755 0.0838 0.0562 0.0133 0.0384 0.0758 0.0336 0.0698 0.109 0.0343 ...
##  $ runspererror: num  17.25 14.5 9 2.68 9.67 ...
##  $ hitspererror: num  38.25 27.75 19.17 5.82 18.78 ...
##  $ hrspererror : num  7.75 4.5 2.833 0.545 0.889 ...
##  $ soserrors   : int  320 276 696 1408 477 445 495 128 78 672 ...
##  $ sbsobp      : num  1.21 0 2.02 6.13 1.04 ...
##  $ sbsruns     : int  276 0 324 1239 261 2288 0 32 560 507 ...
##  $ sbshits     : int  612 0 690 2688 507 3740 0 76 854 832 ...
```

We will try to validate the performance of our model by running it on that baseball dataset on all its predictors, with 'salary' as the response variable. Running our selection model on that dataset with varying values of parameters (such as number of crossover points, generation gap ratio, generation size, selection method, and mutation rates), we consistently received the following results:

```
set.seed(0)
ex <- select(baseball,'salary',cross_point=3,generation_gap = 0.5,gen_size = 40,
             max_iter = 50,selection_mode=2,details = T)
```




```
print(ex$best_cand)
```

```
## [1] "homeruns"    "rbis"        "walks"       "sos"         "freeagent"
## [6] "arbitration" "walksperso"  "sbsobp"
```

```
print(ex$fitness_score)
```

```
## [1] -5375.362
```

```
print(ex$iteration)
```

```
## [1] 31
```

Plugging those predictors into the glm function with the baseball dataset, we see that these predictors have coefficients that are far from 0, which suggests that those fields contribute to the response variable, and that our model is working as intended.

```
glm(data=baseball,formula=salary ~ homeruns + rbis + walks + sos + freeagent +
      arbitration + walksperso + sbsobp)
```

```
##
## Call:  glm(formula = salary ~ homeruns + rbis + walks + sos + freeagent +
##     arbitration + walksperso + sbsobp, data = baseball)
##
## Coefficients:
## (Intercept)     homeruns         rbis        walks          sos    freeagent
##      117.73        27.30        17.69        10.29       -14.20      1294.00
## arbitration   walksperso       sbsobp
##      823.20      -393.22        47.39
##
## Degrees of Freedom: 336 Total (i.e. Null);  328 Residual
## Null Deviance:       516600000
## Residual Deviance: 157300000     AIC: 5375
```

Changing these parameters, we found that slightly increasing the mutation rate from its default value of 0.01 makes the results slightly better but often causes it to reach its max number of iterations, due to its failure to converge. We found that choosing selection_mode to 2, which changes the selection of the parents to fitness/random, worked better than the default, which is choosing both parents according to fitness scores independently.

Also, changing the generation gap from 1 (no individuals from the previous generation are retained in the current generation) to 0.5 (half of the best individuals from the previous generation are retained in the current generation) worked better and improved the runtime of the algorithm since it computes fewer crossovers. In addition, increasing the number of crossover points by *cross_point* improves the algorithm but increases the runtime based on the number of crossover points.

We see that the algorithm generally halts at around the 15-35th generation, but this value varies depending on the *diversity_fact* and *reltol* inputs as well as changing some parameters like *mutate_prob* and *cross_point*. For testing changes to those parameters, we would increase the maximum number of iterations, but it is generally consistent that changes that would worsen the runtime of the algorithm would either improve or not affect its results, while changes that improve its runtime either hinder or not affect its results.

Conclusion

Overall, we have succeeded in creating a genetic algorithm that is able to take in a dataset and response variable and is able to output the list of predictors that is able to fit the response variable to a linear model the best. We have also made that algorithm very flexible by including a lot of features in the algorithm as parameters so the user would have a lot of customization in how they want the algorithm to be ran. These additions to the algorithm would affect its convergence rates, runtime, and best fitness score. We also noticed a tradeoff between the algorithm's runtime and best fitness score.

Contribution

Kevin wrote the code for the main selection function (and its helper page) and some of its helper functions, e.g., functions for breeding like parent-selection, cross-over and mutation. He wrote the implementation part of the report. Hanson wrote the helper function for obtaining the fitness scores of individual binary vectors, and he also contributed to the testing for the main selection function, and the documentation for the datasets. He wrote the result part of the report. Kairui(Nicholas) implemented the **init** function, and he did the testing for the helper functions, including selecting parents, remove_all_zero, mutating a generation, and his own initialization function. He also wrote part of the testing part and also the dummy_data result part in the project report.