

CIS600 HW3

Yuchen Wang
905508464

1. Citations

<https://fluxml.ai/Flux.jl/stable/>

<https://docs.juliaplots.org/latest/tutorial/>

<https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/>

<https://dev.to/dillir07/genetic-algorithm-travelling-salesman-with-julia-477a>

2. Data Set

The data set I chose is about sales data of mobile phones of various companies. The data contains battery_power(Total energy a battery can store in one time measured in mAh); blue(Has bluetooth or not); clock_speed(speed at which microprocessor executes instructions); dual_sim(Has dual sim support or not); fc(Front Camera megapixels); four_g(Has 4G or not); int_memory(Internal Memory in Gigabytes); m_dep (Mobile Depth in cm); mobile_wt(Weight of mobile phone) and price_range(This is the target variable with value of 0(low cost), 1(medium cost), 2(high cost) and 3(very high cost))

2,000 rows × 21 columns (omitted printing of 13 columns)								
	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep
	Int64	Int64	Float64	Int64	Int64	Int64	Int64	Float64
1	842	0	2.2	0	1	0	7	0.6
2	1021	1	0.5	1	0	1	53	0.7
3	563	1	0.5	1	2	1	41	0.9
4	615	1	2.5	0	0	0	10	0.8
5	1821	1	1.2	0	13	1	44	0.6
6	1859	0	0.5	1	3	0	22	0.7
7	1821	0	1.7	0	4	1	10	0.8
8	1954	0	0.5	1	0	0	24	0.8
9	1445	1	0.5	0	0	0	53	0.7
10	509	1	0.6	1	2	1	9	0.1
11	769	1	2.9	1	0	0	9	0.1
12	1520	1	2.2	0	5	1	33	0.5
13	1815	0	2.8	0	2	0	33	0.6
14	803	1	2.1	0	7	0	17	1.0
15	1866	0	0.5	0	13	1	52	0.7
16	775	0	1.0	0	3	0	46	0.7
17	838	0	0.5	0	1	1	13	0.1

Therefore, the data can be treated as a 2-class classification problem with first class of low to medium price range (0 and 1 price range) and second class of high to very high cost (2 and 3 price range).

```
df = file |> Tables.matrix

mat_0 = df[df[:,21] .<= 1, :]
mat_1 = df[df[:,21] .>= 2, :]
```

I use 70% of the data to train the model and use the rest to do the test.

```
train_data = randsubseq(1:1000, 0.7)
train_df = vcat(mat_0[train_data, :], mat_1[train_data, :])

test_data = [i for i in 1:1000 if isempty(searchsorted(train_data, i))]
test_df = vcat(mat_0[test_data, :], mat_1[test_data, :])
```

3. Neural Network

3.1 Generating initial population

```
#build the model using sigmoid as Activation Function
#The input dim is 12 as 12 features and the output dim is 2 represent A and B
function sigmoidModel(batch_size)
    model = Chain(Dense(batch_size, 12, relu), Dense(12, 1, relu))
    return model
end
```

✓ 0.2s

sigmoidModel (generic function with 1 method)

```
# We will define a function that can generate initial weight and bias for the neutral network.
```

```
function generateInitialPopulation(initial_population_size)
    chromosomes = []
    for i in 1 : initial_population_size
        chromosome = sigmoidModel(batch_size)
        push!(chromosomes, chromosome)
    end
    return chromosomes
end
```

✓ 0.2s

generateInitialPopulation (generic function with 1 method)

We define a function that can generate models with initial weights and bias for the neural network.

The models use sigmoidal node functions as the activation function.

```
# basic parameters
batch_size = 20
# population_size = 300
population_size = 30
Mutation_rate = 0.5
recombination_rate = 0.4
Retain_rate = 0.9
iteration = 500
layer = 2

✓ 0.1s
```

Population size in ES is much smaller than GA used in the last homework

3.2 Recombination Function

Create a offspring model by averaging parameters from two chromosome models

```
# Recombination for ES
function recombination(parent_one_chromosome, parent_two_chromosome)
    n_offspring = deepcopy(parent_one_chromosome)
    for i in 1:layer
        select_one_layer = shuffle(collect(1:layer))[1]
        mother_bias = parent_two_chromosome[select_one_layer].bias
        row_to_shuffle = shuffle(collect(1:size(mother_bias)[1]))[1]
        if rand() < recombination_rate
            n_offspring[select_one_layer].bias[row_to_shuffle] = (mother_bias[row_to_shuffle] + n_offspring[select_one_layer].bias[row_to_shuffle]) / 2
        end
    end

    for i in 1:layer
        select_one_layer = shuffle(collect(1:layer))[1]
        mother_weight = parent_two_chromosome[select_one_layer].weight
        row_to_shuffle = shuffle(collect(1:size(mother_weight)[1]))[1]
        col_to_shuffle = shuffle(collect(1:size(mother_weight)[2]))[1]
        if rand() < recombination_rate
            n_offspring[select_one_layer].weight[row_to_shuffle, col_to_shuffle] = (mother_weight[row_to_shuffle, col_to_shuffle] + n_offspring[select_one_layer].weight[row_to_shuffle, col_to_shuffle]) / 2
        end
    end
    return n_offspring
end

✓ 0.1s
```

3.3 Mutation Function

```
# Do mutation here
function mutation(offspring)
    nn = deepcopy(offspring)
    for i in 1:layer
        select_one_layer = shuffle(collect(1:layer))[1]
        bias = offspring[select_one_layer].bias
        row_to_shuffle = shuffle(collect(1:size(bias)[1]))[1]
        if rand() < Mutation_rate
            nn[select_one_layer].bias[row_to_shuffle] += rand(Uniform(-0.5, 0.5), 1)[1]
        end
    end

    for i in 1:layer
        select_one_layer = shuffle(collect(1:layer))[1]
        weight = nn[select_one_layer].weight
        row_to_shuffle = shuffle(collect(1:size(weight)[1]))[1]
        col_to_shuffle = shuffle(collect(1:size(weight)[2]))[1]
        if rand() < Mutation_rate
            nn[select_one_layer].weight[row_to_shuffle, col_to_shuffle] += rand(Uniform(-0.5, 0.5), 1)[1]
        end
    end
    return nn
end

✓ 0.2s
```

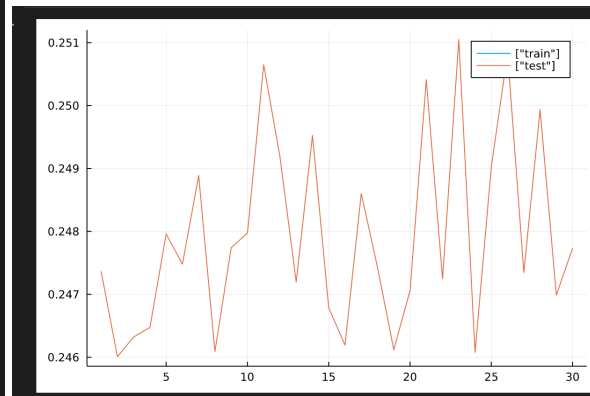
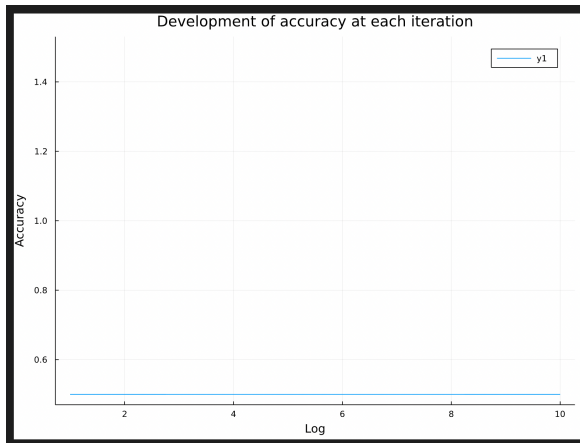
Mutation mechanism I'm using here is random change. The random changes are made on parameters of models.

3.4 Fitness Function

```
accuracy(x, y, i) = mean(vec(chromosomes[i](x) .> 0.5) .== y)
```

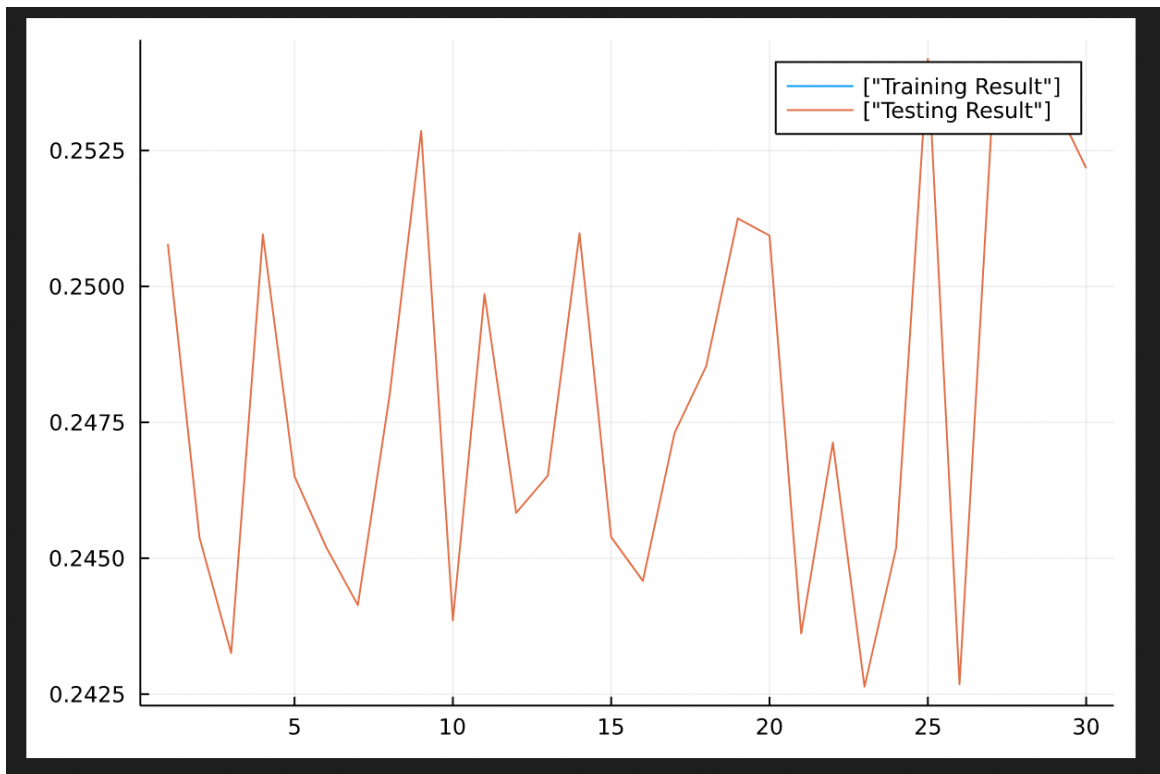
The fitness is used to check how accuracy the model can make a prediction on a dataset x

4 Outcome



The left graph is from the homework one

The right graph is from the last homework, created with GA



The graph above is from this homework, created with ES

Seems the result of ES is a little bit better than GA, and is much better and reasonable than homework one.

5 Confusion Matrix

```
... confusion matrix:  
True True Result: 104, True False Result: 518  
False False Result: 104, False True Result: FT
```

```
confusion matrix:  
True True Result: 354, True False Result: 290  
False False Result: 354, False True Result: FT
```

The confusion matrix at top is from the last homework, created with GA

The confusion matrix at bottom is from this homework, created with ES

There are no many differences between them