

Original Proposal:

Hyperparameter Tuning Through Evolutionary Search

Edward Kim
September 18, 2021

Goals:

Apply evolution on an evolutionary algorithm to minimize the number of generations required to evolve a solution.

Learning Objective:

Explore the application of evolutionary algorithms to human-competitive hyperparameter tuning, through the example of optimizing a genetic algorithm but in the broader context of applications to machine learning.

Milestones:

1. Refactor a genetic algorithm from a top-level control class to an encapsulated object class, to serve as an individual itself to be evolved by a meta-evolutionary algorithm.
2. Create a wrapper class to encode the hyperparameters (mutation rate, population size, selection type, elitism percent, crossover on/off) of a genetic algorithm in a bit string, to represent it as a genotype of an individual.
3. Create methods in the above class to express any given genotype (bit string encoding hyperparameters) to a phenotype (instance of a genetic algorithm), and to evaluate its fitness by running the genetic algorithm and measuring the number of generations taken to evolve a solution.
4. Construct a meta-evolutionary loop, with appropriate classes to represent each generation of genetic-algorithm-individuals and the top-level evolution algorithm.
5. Engineer runtime efficiency to handle the computational cost at multiple orders of magnitude greater than that of a genetic algorithm.
 - a. **Milestone 5 is designated as the minimum “pass”, as a severely time-intensive meta-evolutionary algorithm would not be human-competitive for hyperparameter tuning.**

6. Implement a system, text-based or visual, to display the evolution progress in real time with information on the type of algorithms being evolved (their hyperparameters).
7. Conduct experiments to investigate any interesting trends or behaviors found in the meta-evolution results.

Checkpoints:

1. Create classes to represent a genetic-algorithm-individual, their generation, and their evolution algorithm, and clearly rename all classes to avoid confusion with the genetic algorithm's individual, generation and evolution algorithm classes. (Distinguish the two levels of evolution.)
2. Determine the length and composition of a bit string to encode the hyperparameters of a genetic algorithm, choosing which hyperparameters to use and their order in the bit string (irrelevant to evolution, but relevant to gene expression to phenotype).
3. Create a method in the genetic-algorithm-individual class to decode the bit string genotype to a phenotype, which will be an instance of the genetic algorithm configured with the hyperparameters specified.
4. Create a method in the same class to evaluate the fitness of a genetic algorithm phenotype, by running it and measuring the number of generations required to evolve a solution.
5. Complete the generation and evolution algorithm classes to construct a meta-evolutionary loop.
6. Reduce runtime by minimizing the number of times the fitness() method (described in Checkpoint 4) is called, as it is the most computationally heavy step in the meta-evolutionary loop.
7. Create a system to identify the best and worst genetic algorithm in each generation, and print-out/visualize their fitness value and hyperparameters.
8. Conduct experiments.

Minimum to complete to "pass": Milestone 5, to be tested by Checkpoint 6

Milestone1: Refactor genetic algorithm to an encapsulated object

Time taken: 30 minutes

Notes of how easy/hard it was: easiest

Checkpoint (if applicable) demonstrating success: Checkpoint 1



Classes presented in order individual-generation-algorithm, then again in the next level.

Milestone2: Create wrapper class to encode hyperparameters

Time taken: 2 hours

Notes of how easy/hard it was: hard, due to planning

Checkpoint (if applicable) demonstrating success: Checkpoint 2

```
public class Algo {  
  
    // specify evolutionary objective here (of the GeneticAlgorithm)  
    public String fitnessType = "countNumOf1s"; // or "countNumOf0s", "maxConsec1s", "maxConsec0s"  
  
    /**  
     * terminate at generation 200, those that haven't found a solution will be penalized proportionally  
     * fix chromosome genome length at 100, terminate at chromosome score 95  
     * vary: mutation rate, population size, selection type, elitism percent, crossover on/off  
     */  
    * mutation rate from 0 to 100, encode in a 7-length bit string  
    * handle 101-127 as 0  
    * population size from 10 to 200, encode in a 8-length bit string  
    * handle 0-9, 201-255 as 10  
    * selection type between truncation(0) and roulette wheel(1), encode in 1 bit  
    * elitism percent from 0 to 100, encode in a 7-length bit string  
    * handle 101-127 as 0  
    * crossover on(1) and off(0), encode in 1 bit  
    *  
    * Algo will have genome length 7 + 8 + 1 + 7 + 1 = 24 bits  
    */  
}
```

Definition of the genotype

```
// Fields  
public ArrayList<Integer> genes = new ArrayList<Integer>();  
public Integer length = 24;  
public GeneticAlgorithm algorithm = new GeneticAlgorithm();  
  
// fixed parameters  
public Integer maxGeneration = 200;  
public Integer genomeLength = 100;  
public Integer terminationScore = 95;  
  
// variable parameters  
public Integer mutationRate;  
public Integer populationSize;  
public String selectionType;  
public Integer elitismPercent;  
public Boolean crossover;
```

Parameters of the Algo class, i.e. the hyperparameters of the GeneticAlgorithm

Milestone3.1: Express genotype to phenotype

Time taken: 30 minutes

Notes of how easy/hard it was: easy

Checkpoint (if applicable) demonstrating success: Checkpoint 3

```
// expressPhenotype()
public void expressPhenotype() {

    // instantiate string to combine all genes (binary)
    StringBuilder binaryString = new StringBuilder();

    // transcribe binary genes to string
    for (int i = 0; i < length; i++) {
        binaryString.append(genes.get(i));
    }

    // decode genes to algorithm parameters
    Integer decodedMutationRate = Integer.parseInt(binaryString.substring(0, 7), 2);
    Integer decodedPopulationSize = Integer.parseInt(binaryString.substring(7, 15), 2);
    Boolean decodedIsTruncation = (genes.get(15) == 0);
    Integer decodedElitismPercent = Integer.parseInt(binaryString.substring(16, 23), 2);
    Boolean decodedCrossoverOn = (genes.get(23) == 1);

    // handle exceptional bounds
    if (decodedMutationRate > 100) {
        decodedMutationRate = 0;
    }
    if (decodedPopulationSize % 2 == 1) {
        decodedPopulationSize++;
    }
    if (decodedPopulationSize < 10 || decodedPopulationSize > 200) {
        decodedPopulationSize = 10;
    }

    if (decodedElitismPercent > 100) {
        decodedElitismPercent = 0;
    }

    // express genes to parameters, to control GeneticAlgorithm in the fitness() method below
    mutationRate = decodedMutationRate;
    populationSize = decodedPopulationSize;
    if (decodedIsTruncation) {
        selectionType = "Truncation";
    } else {
        selectionType = "Roulette Wheel";
    }
    elitismPercent = decodedElitismPercent;
    crossover = decodedCrossoverOn;
} // end method
```

The method extracts and converts all binary values of the hyperparameters from the bit string to decimal numbers, handles exceptions, and saves them as a field in the Algo class to configure an instance of the GeneticAlgorithm (also a field in the Algo class).

Milestone3.2: Evaluate fitness of phenotype

Time taken: 2 hours

Notes of how easy/hard it was: hard, due to debugging

Checkpoint (if applicable) demonstrating success: Checkpoint 4

```
source_code > Algo.java > Algo > Algo(ArrayList<Integer>)
29     public static void main(String[] args) {
30
31         for (int i = 0; i < 20; i++) {
32             new Algo().fitness();
33         }
34     }
```

PROBLEMS 2 OUTPUT TERMINAL DEBUG CONSOLE

ALGORITHM FOUND SOLUTION AT GENERATION 308
ALGORITHM FOUND SOLUTION AT GENERATION 293
ALGORITHM FOUND SOLUTION AT GENERATION 296
ALGORITHM FOUND SOLUTION AT GENERATION 302
ALGORITHM FOUND SOLUTION AT GENERATION 299
ALGORITHM FOUND SOLUTION AT GENERATION 326
ALGORITHM FOUND SOLUTION AT GENERATION 293
ALGORITHM FOUND SOLUTION AT GENERATION 287
ALGORITHM FOUND SOLUTION AT GENERATION 284
ALGORITHM FOUND SOLUTION AT GENERATION 278
ALGORITHM FOUND SOLUTION AT GENERATION 305
ALGORITHM FOUND SOLUTION AT GENERATION 314
ALGORITHM FOUND SOLUTION AT GENERATION 275

The Algo individuals to the left are randomly generated, which accounts for the relatively high (bad) fitness values.

Milestone4: Construct meta-evolutionary loop

Time taken: 30 minutes

Notes of how easy/hard it was: easy

Checkpoint (if applicable) demonstrating success: Checkpoint 5

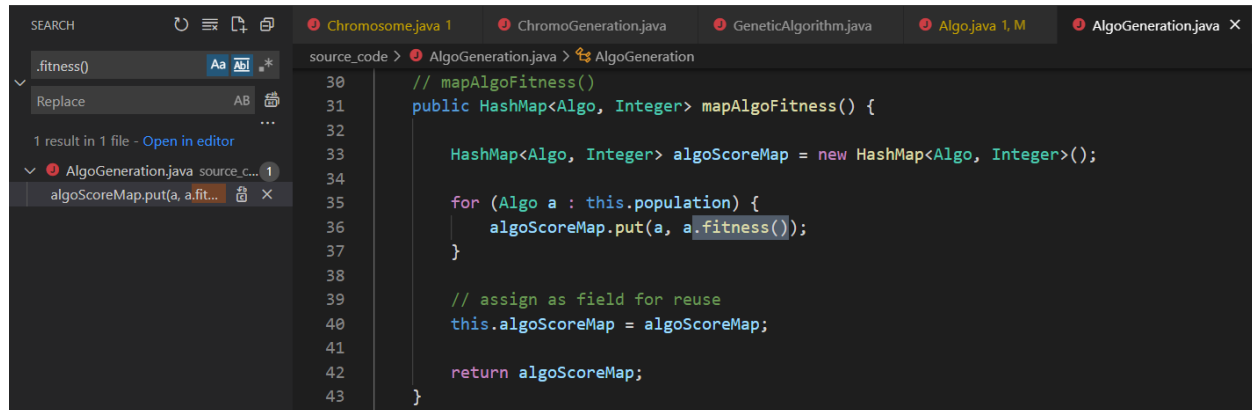
```
source_code > MetaAlgorithm.java > ...
37     // Fields
38     ArrayList<AlgoGeneration> allGenerations = new ArrayList<AlgoGeneration>();
39
40
41     // createGeneration method
42 > public void createGeneration(Integer genNumber, Integer populationSize) { ...
84
85
86     // nextGeneration method
87 > public void nextGeneration(Integer genNumber, Integer mutationRate) { ...
343
```

Milestone5: Reduce runtime

Time taken: 3 hours

Notes of how easy/hard it was: hardest, due to significant refactoring

Checkpoint (if applicable) demonstrating success: Checkpoint 6



```
source_code > Chromosome.java 1 ChromoGeneration.java GeneticAlgorithm.java Algo.java 1, M AlgoGeneration.java X
SEARCH
Replace
1 result in 1 file - Open in editor
AlgoGeneration.java source_c... 1
algoScoreMap.put(a, a.fitness())
30 // mapAlgoFitness()
31 public HashMap<Algo, Integer> mapAlgoFitness() {
32
33     HashMap<Algo, Integer> algoScoreMap = new HashMap<Algo, Integer>();
34
35     for (Algo a : this.population) {
36         algoScoreMap.put(a, a.fitness());
37     }
38
39     // assign as field for reuse
40     this.algoScoreMap = algoScoreMap;
41
42     return algoScoreMap;
43 }
```

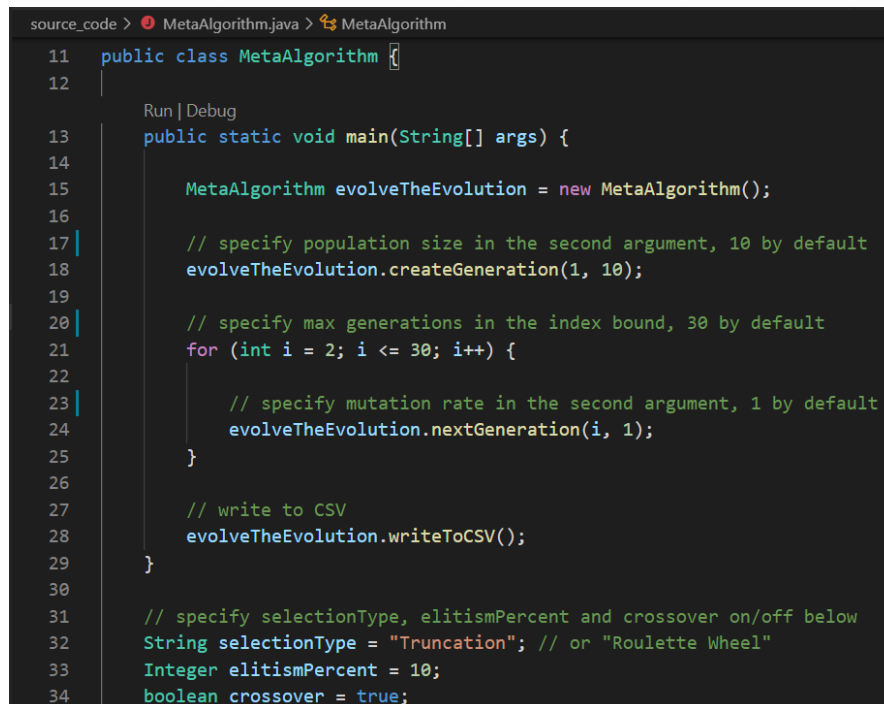
The fitness() method is called only once in the entire repository, when an algoScoreMap is created for each generation to be reused for all purposes regarding fitness scores.

Milestone6: Display evolution progress

Time taken: 1 hour

Notes of how easy/hard it was: medium

Checkpoint (if applicable) demonstrating success: Checkpoint 7



```
source_code > MetaAlgorithm.java X MetaAlgorithm
11 public class MetaAlgorithm {
12
13     Run | Debug
14     public static void main(String[] args) {
15
16         MetaAlgorithm evolveTheEvolution = new MetaAlgorithm();
17
18         // specify population size in the second argument, 10 by default
19         evolveTheEvolution.createGeneration(1, 10);
20
21         // specify max generations in the index bound, 30 by default
22         for (int i = 2; i <= 30; i++) {
23
24             // specify mutation rate in the second argument, 1 by default
25             evolveTheEvolution.nextGeneration(i, 1);
26         }
27
28         // write to CSV
29         evolveTheEvolution.writeToCSV();
30
31         // specify selectionType, elitismPercent and crossover on/off below
32         String selectionType = "Truncation"; // or "Roulette Wheel"
33         Integer elitismPercent = 10;
34         boolean crossover = true;
```

The method to run.

```

Generation 1:
The best algorithm evolved a solution in 269 generations, with mutation rate 23, population size 192, selection type Roulette Wheel, 15% elitism, and crossover off
The worst algorithm evolved a solution in 326 generations, with mutation rate 98, population size 36, selection type Roulette Wheel, 0% elitism, and crossover off

    5 crossovers complete
    Truncation successful with 1 elite children

Generation 2:
The best algorithm evolved a solution in 272 generations, with mutation rate 11, population size 200, selection type Roulette Wheel, 53% elitism, and crossover off
The worst algorithm evolved a solution in 299 generations, with mutation rate 24, population size 10, selection type Truncation, 92% elitism, and crossover off

    5 crossovers complete
    Truncation successful with 1 elite children

Generation 3:
The best algorithm evolved a solution in 263 generations, with mutation rate 28, population size 48, selection type Roulette Wheel, 15% elitism, and crossover off
The worst algorithm evolved a solution in 293 generations, with mutation rate 28, population size 46, selection type Truncation, 92% elitism, and crossover on

    5 crossovers complete
    Truncation successful with 1 elite children

Generation 4:
The best algorithm evolved a solution in 266 generations, with mutation rate 5, population size 156, selection type Roulette Wheel, 52% elitism, and crossover off
The worst algorithm evolved a solution in 284 generations, with mutation rate 51, population size 84, selection type Roulette Wheel, 52% elitism, and crossover off

    5 crossovers complete
    Truncation successful with 1 elite children

Generation 5:
The best algorithm evolved a solution in 254 generations, with mutation rate 3, population size 200, selection type Roulette Wheel, 53% elitism, and crossover off
The worst algorithm evolved a solution in 290 generations, with mutation rate 11, population size 10, selection type Roulette Wheel, 53% elitism, and crossover off

    5 crossovers complete
    Truncation successful with 1 elite children

Generation 6:
The best algorithm evolved a solution in 257 generations, with mutation rate 3, population size 200, selection type Roulette Wheel, 53% elitism, and crossover off
The worst algorithm evolved a solution in 299 generations, with mutation rate 75, population size 200, selection type Roulette Wheel, 0% elitism, and crossover off

    5 crossovers complete
    Truncation successful with 1 elite children

```

...

```

ALGORITHM FOUND SOLUTION AT GENERATION 47

ALGORITHM FOUND SOLUTION AT GENERATION 153

ALGORITHM FOUND SOLUTION AT GENERATION 44

ALGORITHM FOUND SOLUTION AT GENERATION 117

ALGORITHM FOUND SOLUTION AT GENERATION 146

Generation 29:
The best algorithm evolved a solution in 44 generations, with mutation rate 1, population size 170, selection type Truncation, 0% elitism, and crossover on
The worst algorithm evolved a solution in 221 generations, with mutation rate 1, population size 176, selection type Truncation, 31% elitism, and crossover off

    5 crossovers complete
    Truncation successful with 1 elite children

ALGORITHM FOUND SOLUTION AT GENERATION 50

ALGORITHM FOUND SOLUTION AT GENERATION 54

ALGORITHM FOUND SOLUTION AT GENERATION 45

ALGORITHM FOUND SOLUTION AT GENERATION 190

ALGORITHM FOUND SOLUTION AT GENERATION 44

ALGORITHM FOUND SOLUTION AT GENERATION 58

ALGORITHM FOUND SOLUTION AT GENERATION 130

ALGORITHM FOUND SOLUTION AT GENERATION 141

Generation 30:
The best algorithm evolved a solution in 44 generations, with mutation rate 1, population size 170, selection type Truncation, 0% elitism, and crossover on
The worst algorithm evolved a solution in 296 generations, with mutation rate 33, population size 172, selection type Truncation, 0% elitism, and crossover on

Evolution data successfully written to plot_data.csv

```

Console print-outs describing the fitness and hyperparameters of the best and worst algorithms in each generation, as well as the occasional success of each algorithm in evolving a solution under 200 generations (much more frequent towards the end of the meta-evolution).

Milestone7: Experiments

Time taken: 1 hour

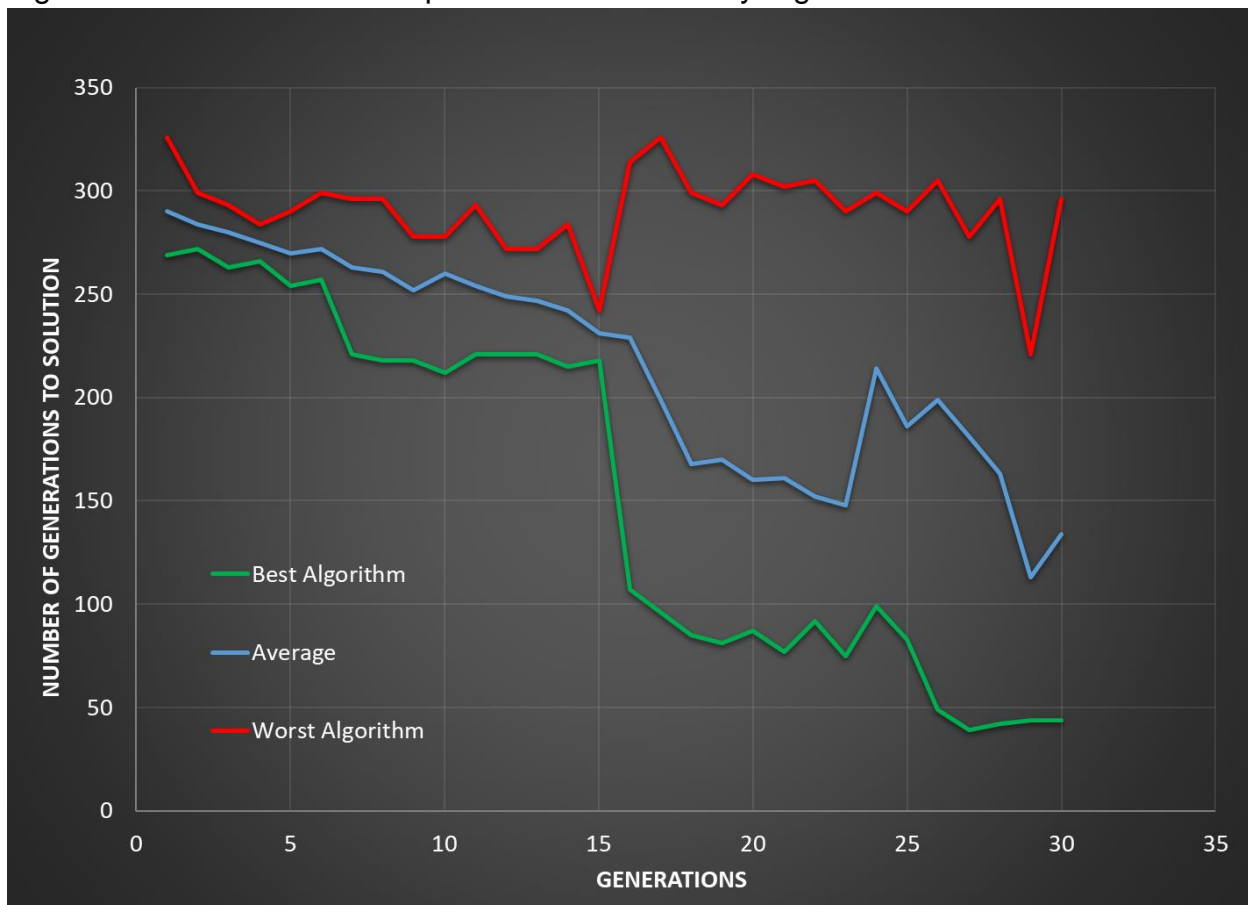
Notes of how easy/hard it was: easy

Checkpoint (if applicable) demonstrating success: Checkpoint 8

Setup:

Meta-Evolution Hyperparameter	Value
Population Size	10
Max Generations	30
Mutation Rate	1
Selection Method	Truncation
Elitism	10%
Crossover	On

Figure 1. Meta-Evolution to Optimize an Evolutionary Algorithm



In the last generation of the meta-evolution, the best algorithm evolved a solution in 44 generations, with mutation rate 1, population size 170, selection type Truncation, 0% elitism, and crossover on.

From Figure 1, it can be seen that the number of generations required to evolve a solution generally decreases as the meta-evolution proceeds. The occasional decrease in the best algorithm's fitness even with elitism, as well as the decrease in average fitness at generation 24, can all be attributed to the nondeterministic nature of evaluating each genetic algorithm's fitness, which can in itself vary randomly regardless of the "quality" of the chosen hyperparameters. Therefore, it is to be noted that the fitness evaluation could have been more accurate if multiple trials were run on each genetic algorithm and the average number of generations to solution was calculated (though the computational cost would proportionally increase).

It is interesting to observe the distinct increase in diversity at generation 15 and the following appearance of much better algorithms. Examining the console print-outs and the CSV data (all included in Data_Meta_Evolution.xlsx), generation 15 is found to be a transformational point where algorithms using Truncation appear for the first time, then quickly dominate the population with their superior fitness (see screenshot below). At generation 16, one of the first algorithms using Truncation successfully evolves a solution before 200 generations, and rises as the new best algorithm.

```
Generation 14:
The best algorithm evolved a solution in 215 generations, with mutation rate 2, population size 194, selection type Roulette Wheel, 3% elitism, and crossover off
The worst algorithm evolved a solution in 284 generations, with mutation rate 67, population size 70, selection type Roulette Wheel, 36% elitism, and crossover off

5 crossovers complete
Truncation successful with 1 elite children

Generation 15:
The best algorithm evolved a solution in 218 generations, with mutation rate 2, population size 194, selection type Roulette Wheel, 3% elitism, and crossover off
The worst algorithm evolved a solution in 242 generations, with mutation rate 1, population size 194, selection type Truncation, 51% elitism, and crossover off

5 crossovers complete
Truncation successful with 1 elite children

ALGORITHM FOUND SOLUTION AT GENERATION 107

Generation 16:
The best algorithm evolved a solution in 107 generations, with mutation rate 1, population size 194, selection type Truncation, 0% elitism, and crossover off
The worst algorithm evolved a solution in 314 generations, with mutation rate 1, population size 10, selection type Roulette Wheel, 97% elitism, and crossover off
```

Overall, the meta-evolution's solution approaches an expected set of hyperparameters from manually testing the genetic algorithm: mutation rate 1, maximum possible population size, selection by Truncation, 0% or very little elitism, and crossover on. The final solution in this experiment — mutation rate 1, population size 170, selection type Truncation, 0% elitism, and crossover on — has successfully evolved a solution in 44 generations, which is a surprisingly high performance for the genetic algorithm used (compared to when it was manually tuned for higher performance). This demonstrates the effectiveness of the meta-evolution in optimizing the genetic algorithm, and opens the possibility that it might be able to find a novel solution if ran with a bigger population and for more generations.

Self-Evaluation

If you were to assign yourself a score from 1-10 on how well you performed on this assignment, taking into account the challenges you faced, what would you assign yourself? Why?

I would assign a 9, as I satisfactorily realized my initial vision in proposing this project, calmly resolved unexpected challenges (long debugging and planning difficulties), and generated and analyzed interesting results with experimentation. I would reserve a 10 for when I would have implemented all areas for improvement that I have identified, such as averaging the fitness evaluation to tolerate the nondeterministic results.

Would you say that you achieved your Learning Objective? Why?

I have achieved my Learning Objective, as I thoroughly explored making an hyperparameter-tuning evolutionary algorithm human-competitive, both in performance and computational resources. Apparently, hyperparameter evolution — evolutionary algorithms for hyperparameter optimization — is a real technique in the machine learning industry. I did not know that when I proposed this project, but now I fully understand its motives, performance, and challenges for future developments.