# Randomized Optimization Methods on Common Optimization Problems and Neural Net Weighting

Nicholas C. Park - October 6, 2020

## 1. Introduction

Many of the supervised learning algorithms depend on optimization which aims to obtain the state that outputs the minimum underlying cost or maximum fitness. Here the different techniques of optimization are observed and applied to four different problems, or fitness functions: Continuous Peak, Knapsack, Flipflop, and finally in NeuralNet weighting (MLPClassifier) for the classification of poisonous mushrooms via the mushroom dataset.

Similar to the supervised learning algorithms that require parameter tuning using GridSearch via cross-validations of all possible combinations of the learner's parameters, the optimization algorithms also have parameters that affect their performance in terms of how soon, in iteration and in cumulative compute time, they converge (i.e., stay unimproved at their highest function values) and also in terms of their willingness to randomly explore and exploit in the attempt to sight and converge to the global optimum. Some common random optimization/search algorithms (Randomized Hill Climbing, Simulated Annealing, Genetic Algorithm and MIMIC) are implemented in discrete parameter spaces, and their hyperparameters, tuned, according to the varying optimization problems. The first three methods are later also used to optimize the weights of a Neural Net to compare each of their performance against the backpropagation via gradient descent.

## 2. MLROSe-hiive

The optimization problems (including the neural net weight optimization) and the optimization algorithms discussed hereon are implemented by a locally modified version of MLROSe-hiive, whose original version is extended from MLROSe (Machine Learning, Randomized Optimization and Search), a repository constructed to experiment the application of some common randomized optimization and search algorithms to various optimization problems developed by Genevieve Hayes. Modifications to module for this analysis include:

- Changed the four randomized optimizer functions in rhc.py, sa.py, ga.py and mimic.py located in \mlrose\mlrose_hiive\algorithms\ to have a fourth return value: cumulative runtime per iteration as a list.
- Changed the gradient_descent_original() optimizer function located in \mlrose\mlrose_hiive\neural\utils\weights.py to have a fourth return value: cumulative runtime per iteration as a list.
- Changed the gradient_descent_original() optimizer function located in \mlrose\mlrose_hiive\neural\utils\weights.py to append problem.get_adjusted_fitness() instead of problem.get_fitness() to the fitness curve list to output proper sign of loss curve.
- Copied in the eval_node_probs to discrete_opt.py from the original MLROSe into its extended version MLROSe-hiive to enable vectorization of MIMIC by including in the algorithm the feature, mimic_speed.

## 3. Randomized Search Algorithms and their Parameters

### 3.1 Randomized Hill Climbing

The conventional hill climbing is an algorithm that starts from a random solution and locally searches for a better solution by incremental change until no improvement is made within max_attempts specified. Randomized hill climbing iteratively climbs the hill with random restarts, a certain number of times, per iteration within the specified max_iters until the best state is found.

### 3.2 Simulated Annealing

At each iteration the Simulated Annealing searches locally like hill climbing but occasionally accepts a poorer fitness function value so that it does not commit to a local optimum and has a chance to escape it. The initial temperature is set by init_temp and it is decreased each iteration according to a schedule. A high temperature accepts worse solution more allowing exploration and gradually the algorithm is set to exploit more than explore. There are three different cooldown (temperature decay) schedule for the simulated annealing with their corresponding decay rates: Arithmetic: $T(t) = \max(T_0 - rt , T_{min})$, Exponential: $T(t) = \max(T_0 e^{-rt}, T_{min})$ and Geometric: $T(t) = \max(T_0 r^t, T_{min})$

### 3.3 Genetic Algorithm

Genetic Algorithms iterate to improve a population of states, or individuals, by producing offspring via crossover from two individuals in it based on the percent to breed specified by pop_breed_percent with a certain mutation probability, mutation_prob, and by iteratively keeping the fittest portion of the population. The population size initially is set by the pop_size and it improves every iteration, from which the best solution is also selected. Iteration is until the best solution becomes optimal solution depending on how long it has not improved whose limit is specified by max_attempts.

### 3.4 MIMIC (Mutual Information Maximizing Input Clustering)

MIMIC randomly samples regions of the input space from the population (initially set by pop_size) most likely to contain the optimum based on the probability density estimated and uses the density estimator to capture the possible structure of the input space. It searches for the optimum by sampling and keeping a portion of the samples each iteration, whose amount is specified by keep_pct.

## 4. Optimization Problems

The problems are within discrete state space and the fitness functions they wrap are something to maximize in the following three optimization problems.

4.1 Parameter Tuning and Performance Comparison using Fitness Curve Plots

The fitness curves are plotted with confidence bands using standard deviation using five different seeds value 0, 10, 20, 30 and 40 during the final comparison of the four randomized optimization algorithms against one another. During parameter tuning, the fitness curves are plotted without confidence bands to showcase exemplary attempts to discuss the effects of varying one parameter in the optimizer algorithm on the fitness curve of the algorithm. The combinations of hyperparameters are measured of their runtime and plotted of their fitness function values against the iterations in a nested for loop. The values are cut off if they are converged according to the below criteria:

- Have reached the global optimum, or the true answer.
- Have stayed unimproved within the per-algorithm specified max_attempts value, i.e. expiring due to no improvement.

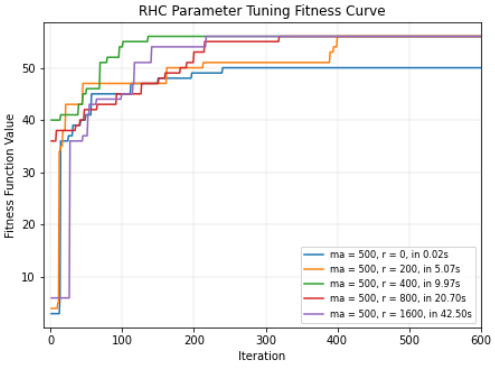
All things equal the parameter combination with the less wall clock time is chosen. The underlying algorithms of the MLROSe-hiive optimizers reveal that the iterations exit either when the max_iters is reached or when the fitness function value does not improve within the specified max_attempts, The max_attempts parameter varies per randomized optimization algorithm used since the notion of iterations is different across the algorithms. It is tuned since it needs to be large enough that, for a problem, one of the parameter combinations of a certain algorithm can reach the global optima without prematurely terminating. It is also not set too high to prevent wasting computation time without improving the fitness function. Therefore, the max_attempts is tuned bottom-top during parameter tuning to obtain the global optimum and later trimmed to save computation time during comparison.

The multiple plots for the following problems are generated by the fitness curve lists saved in .npy object files as 2d numpy arrays to which every run's result is appended. Various colormaps and linestyles from matplotlib has been applied to visualize the effects of varying one hyperparameter on the fitness curves of the algorithm and the selected hyperparameter set has been confirmed of its identity by viewing it from its corresponding saved .npy objects.

4.2 Continuous Peaks

In the Continuous Peaks problem, the optimizer seeks to search for the global optimum in the presence of many local optima in a 1-dimensional space. The underlying fitness function to maximize is as follows: $Fitness(x, T) = \max(max\_run(0, x), max\_run(1, x)) + R(x, T)$ where is $max_run(b, x)$the length of the maximum run of the bit type b (in this case, b = 0 or 1). The problem uses a 30-bit input space, allowing a total of $2^{30}$ possible states. The problem includes a threshold parameter (T) which in this case will be a product of the length of the states (30) and the default value of $t_{cpt}$ (0.1), or T = 3. The global optimum by function value is 57.

**Parameter tuning for Randomized Hill Climbing and MIMIC**




The number of restarts is set from 0 to 1, and doubled up to the value of 256 then doubling the restart starting 200 has shown that restart = 1600 does not converge to local optimum faster than restart = 400 and 800. Global optimum is not reached.

The MIMIC performs best on pop_size = 500 with keep_pct = 0.1 and reaches the global optimum in the 14th iteration. Each parameter has shown no clear relationship with the ability to converge fast.

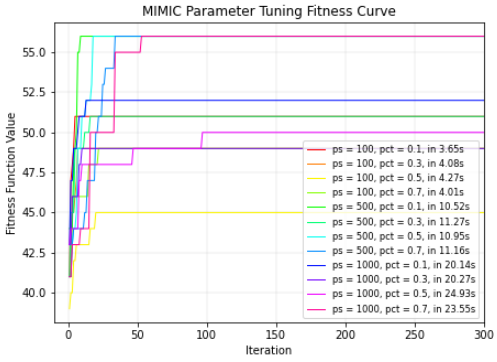
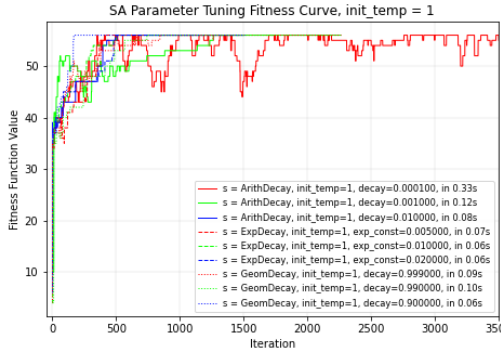



Figure 1 (left) and Figure 2 (right)

**Parameter tuning for Simulated Annealing**

The default max_attempts of 10 shows that all algorithms terminated prematurely before convergence. It is eventually adjusted to 2000 which covers the windows for all possible hyperparameter combinations.



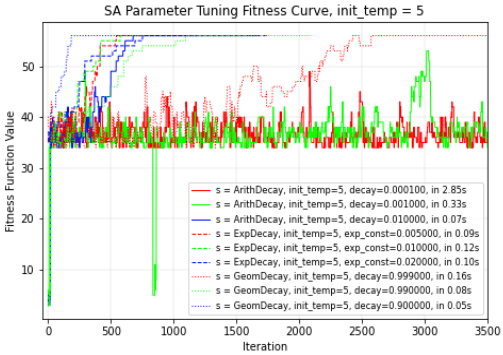





Figure 3

In all cases the geometric decay with the decay rate of 0.9 is the fastest converging to the global optimum. Over reruns, it has shown T = 5 has performed consistently the best amongst all other potential parameter combinations. The exponential decay schedule with the corresponding rate of 0.02 noted by dashed greed lines converge earlier than any other decay schedules with their corresponding rates.

**Parameter Tuning for Genetic Algorithms**

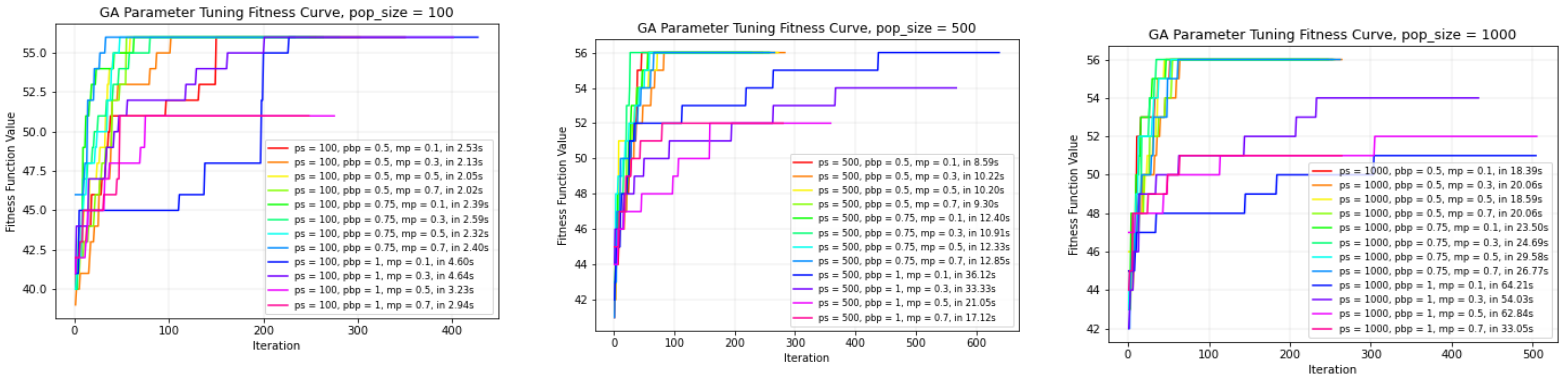



Figure 4

The max_attempts = 200 is found to be enough to filter out the worst performing parameter sets. It turns out that the genetic algorithms population size increase does not necessarily result in the faster convergence for the Continuous Peaks. Increasing the pop_size parameter from 100 to 500 and 1000 has shown a clear linear increase in runtime but has not shown a clear decrease in the number of iteration it takes to converge. The mutation_prob seems to facilitate exploration from the unwillingness converge the local optima in the earlier iterations. this example the pop_breed_percent value of 0.5 is found to be adequate while allowing 0.1 mutation within 100 population size. None of them reached the global optima within max_iters = 2000.

**Selected Parameters**

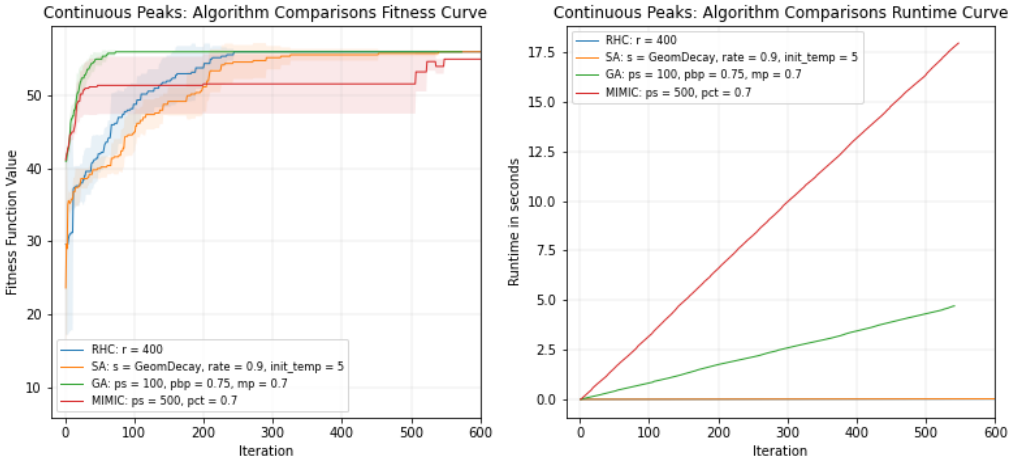
| Algorithm | Max Attempts | Restarts | Average Runtime |
|---|---|---|---|
| RHC | 500 | 0, 200, 400, 800, 1600 | 9.97 seconds |

| Algorithm | Max Attempts | Init Temp | Schedule | Decay Rate | Average Runtime |
|---|---|---|---|---|---|
| Simulated Annealing | 1000 | 1, 5, 10 | Arithmetic, Exponential, Geometric | Arithmetic = [0.0001,0.001,0.01], Exponential = [0.005, 0.01 , 0.02], Geometric = [0.999, 0.99, 0.9]] | 0.05 seconds |

| Algorithm | Max Attempts | Pop_size | pop_breed_percent | Mutation_prob | Average Runtime |
|---|---|---|---|---|---|
| GA | 200 | 100, 200,400,500,800,1000 | 0.25, 0.5, 0.75, 1 | 0.1, 0.3, 0.5 ,0.7 | 2.40 seconds |

| Algorithm | Max Attempts | Pop_size | keep_pct | Average Runtime |
|---|---|---|---|---|
| MIMIC | 300 | 100, **500** ,1000, 2000 | **0.1**, 0.3, 0.5, 0.7 | 10.52 seconds |

**Algorithms Comparison: Mean Fitness Curve and Computation Times**




| Algorithm | Runtime when max_attempt = 1000 |
|---|---|
| RHC | 1.56842s |
| SA | 1.14512s |
| GA | 4.98943s |
| MIMIC | No Convergence |

Figure 5 (on the left)

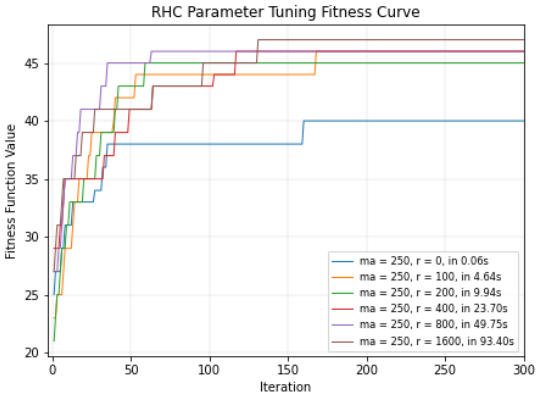
The value of the mutation probability on all the problem also neither as a clear direct nor an inverse relationship on how early it can converge since too much exploration or too much exploitation can both hurt. Also, from the varying values of percentage of population to breed, it shows that too high or too low of a percentage bred results in slower convergence. Taking too much population to breed risks breeding the non-fit individuals. Taking too little population to breed risks breeding less of the fit individuals. Five runs averaged with standard deviation bands show that Genetic Algorithms converge at the smallest iteration count, but not necessarily the earliest in wall clock time. It can be shown that each of the computations time per iteration is approximately linear for all four algorithms in this problem with varying slope. The cumulative computation time shows that iterations for the MIMIC takes the greatest amount of time per iteration, followed by Genetic Algorithm, Randomized Hill Climbing then Simulated Annealing.

## 4.1 Flipflop

Flipflop counts the given bit string's bit alternation. An optimal bit string consists entirely of alternating digits, i.e. 1-0-1-0-1-0 and so on. The input space is of length 50 binary bit string. Therefore, the global optimum of the fitness function is 49.

### 4.1.1 Randomized Hill Climbing

**Parameter tuning for Randomized Hill Climbing and MIMIC**




For RHC, increasing the number of restart results in convergence in less number of iterations at the expense of more wall clock time. It is shown that the restart = 200 also converges in relatively less number of iterations second to the restart = 1600. The randomized hill climbing has trouble locating the global optimum and converges to a local optimum of 47.

In the case of MIMIC shown on right. MIMIC has trouble converging to the global optimum and all parameter combinations get trapped in a local optimum of 47.

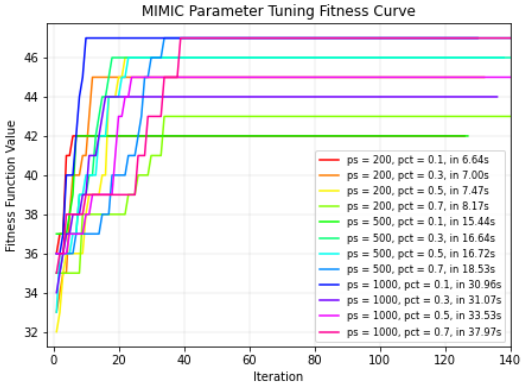
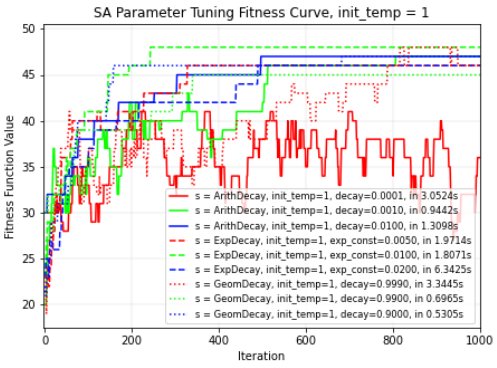
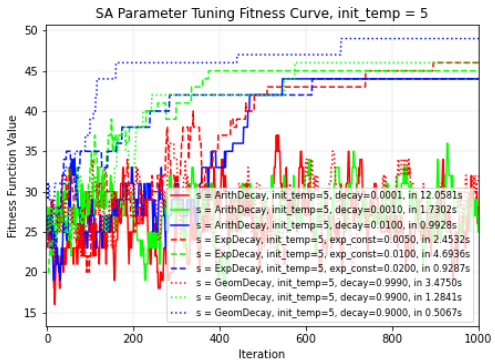



Figure 6 (on the right) and Figure 7 (on the left)

**Parameter Tuning for Simulated Annealing**

The max_attempts has been set high since the simulated annealing has shown to explore more than exploit relative to the given max_attempts on the slow cooling (slow temperature decaying) causing some of the plots terminate prematurely. The averages show that the plots have terminated before having the chance to reach the global optimum that other averages have reached.



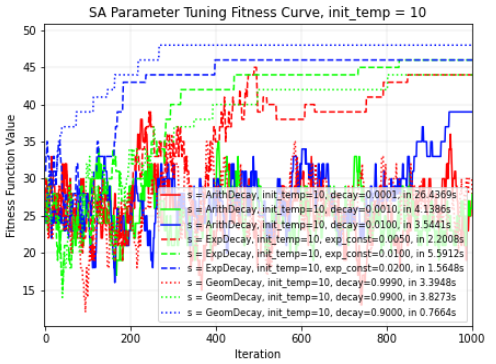




Figure 8

The arithmetic with the slowest decay shows high exploration on figure 8 showing a prolonged zigzagging which suggests that it is exploring excessively due to that its decay is the slowest. The temperature stays too hot for many iterations due to the mathematical nature of this decay and facilitates unnecessary explorations. The same can be said about other solid colored plots which use the arithmetic decay because no matter the rate of this decay

rate, the two other types, i.e. geometric and exponential, decay much faster in the longer run allowing a "correct" cooling. The earliest iterative convergence was obtained by init-temp= 10 with gradual exponential decay using 0.005 decay rate suggesting the cool-off at the right pace.

**Parameter Tuning for Genetic Algorithms**

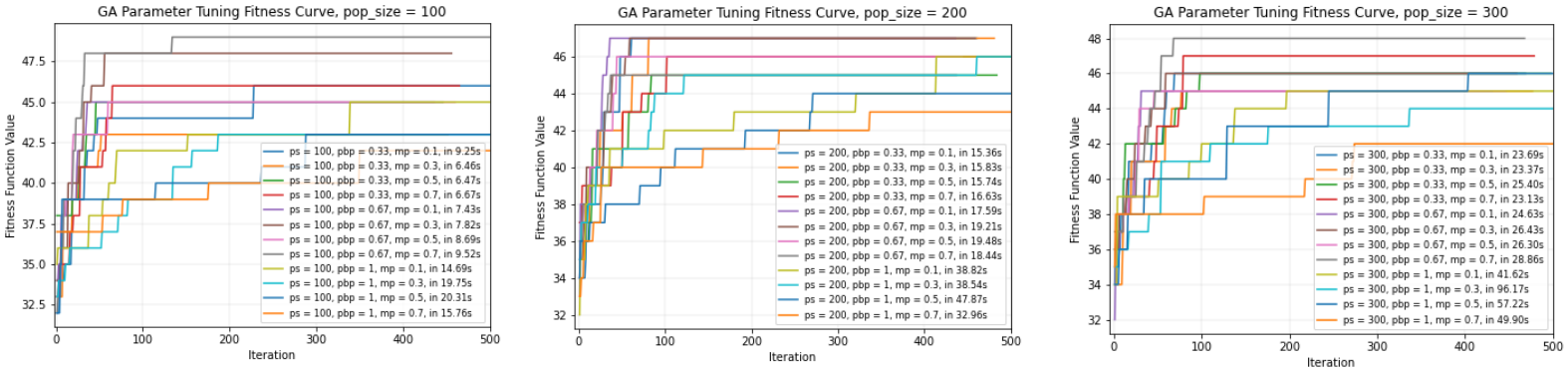



Figure 9

Unlike the continuous peaks or the knapsack problem, the genetic algorithm has benefited from having a large population size. The default population percentage to breed is chosen with very high chance of mutation. And this has converged in the least amount of iterations except it has taken the longest in terms of wall clock time due to high population size. More breeding and mutation implies more computation time, and therefore the pop_size of 100 is chosen.

**Selected Parameters**

| Algorithm | Max Attempts | Restarts | Average Runtime |
|---|---|---|---|
| Randomized Hill Climb | 250 | 0, 200, ,400,800,**1600** | 49.75 seconds |

| Algorithm | max-attempts | Init Temp | Form | Decay Rate | Average Runtime |
|---|---|---|---|---|---|
| Simulated Annealing | 8000 | 1, 5, **10** | Arithmetic, **Exponential**, Geometric | Arithmetic = [0.0001,0.001,0.01], Exponential = [**0.005**, 0.01 , 0.02], Geometric = [0.999, 0.99, 0.9]] | 0.5067 seconds |

| Algorithm | max_attempts | Pop_size | pop_breed_percent | Mutation_prob | Average Runtime |
|---|---|---|---|---|---|
| Genetic Algorithms | 300 | 100, 200,400,500,800,**1000** | 0.33, 0.67, 1 | 0.1, 0.3, 0.5 ,**0.7** | 9.52 seconds |

| Algorithm | max_attempts | Pop_size | Pct_kept | Average Runtime |
|---|---|---|---|---|
| MIMIC | 300 | 200,500 ,**1000** | 0.1, 0.3, 0.5 ,**0.7** | 18.53 seconds |

**Algorithms Comparison: Mean Fitness Curve and Computation Times**

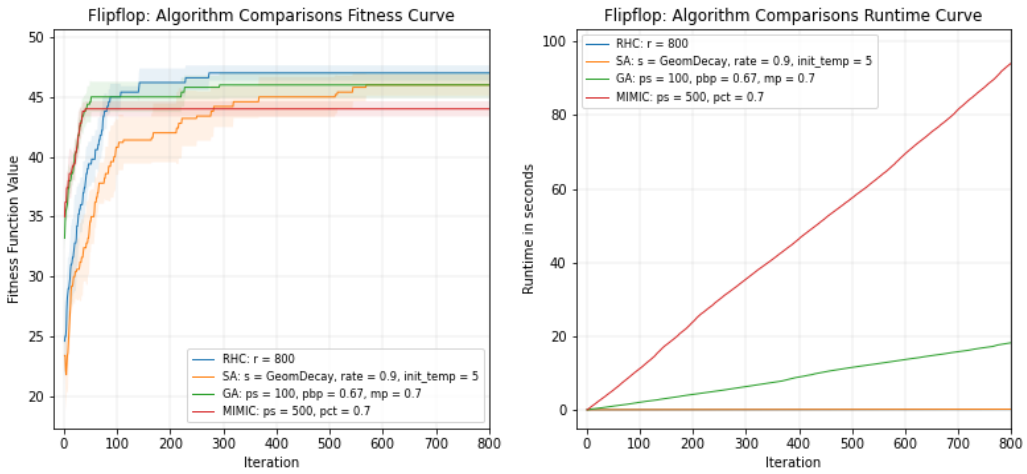



Figure 10

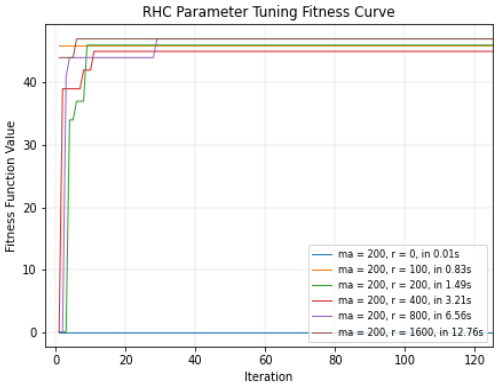
In MIMIC, the population size and percentage kept both result in increased runtime, when all other things are equal. The GA is the only algorithm that reached the global optimum. All other algorithm converged to the local minimum given max_attempt = 1000. Simulated Annealing has a chance of outperforming GA due to its fast iteration speed.

| Algorithm | max_attempt = 1000 runtime |
|---|---|
| RHC | 2.78942s |
| SA | (No Convergence) |
| GA | (No Convergence) |
| MIMIC | (No Convergence) |

3.2 The Knapsack problem

The knapsack problem maximizes the total value of a combinatorial collection of items (with replacement) from a list of items, each given a specific value and weight, given the restraint that the total weight must be less than specified. The following 9 items are introduced in the respective order: weights = [10, 5, 2, 8, 15, 7, 7, 7, 7], values = [1, 2, 3, 4, 5, 6, 7, 8, 9] and max_weight_pct = 0.6 (i.e., the restraint). The global optimum is 48 via having 16 of the same third item which has weight = 2 and value = 3.

**Parameter Tuning for Random Hill Climb**




The purple line (restart = 800) converges to the local optima earliest given max_attempt = 100 (initially set to 1000 then sliced off) at half the time of brown (restart = 1600). Number of restarts directly proportional to convergence time. None reached the global optimum.

For MIMIC on Figure 12, population size of 80 and percentage kept of 0.1 converge to the local optimum earliest. In this case also, none reached the global optimum.

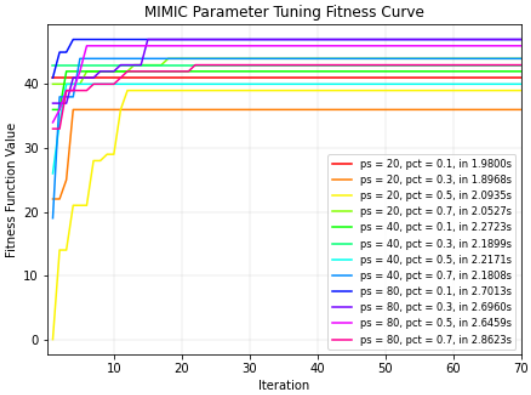
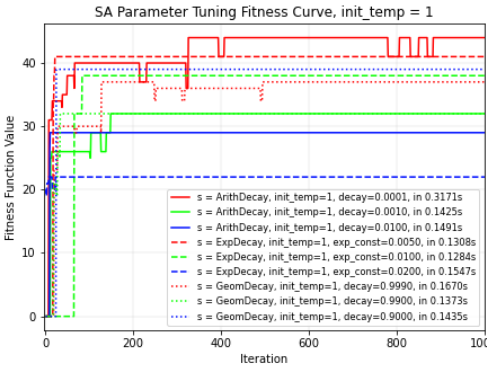
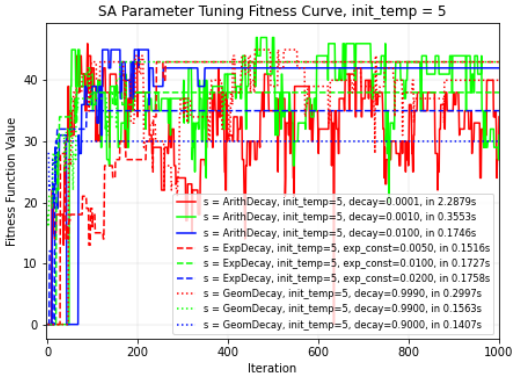



Figure 11 (on the left) and Figure 12 (on the right)

**Parameter Tuning for Simulated Annealing**



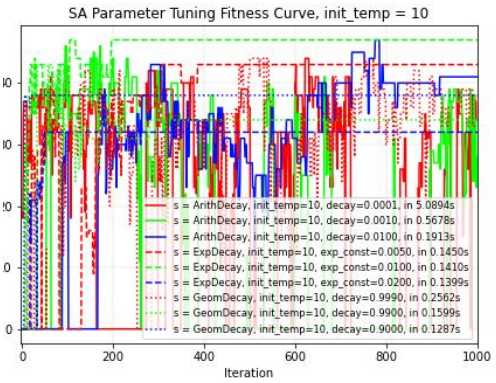




Figure 13

init_temp = 1 fails to converge to the global optimum for all algorithms due to that it starts off “cool” and does not have much chance to explore to other basins of attraction. It is shown that higher values of init_temp when adjusted from 1 to 5 and 5 to 10 facilitates too much exploration which causes the zigzag pattern to occur more frequently. The blue lines with the fastest decay tend to do well in all init_temp settings converging relatively quickly in iteration and runtime. This is likely because the corresponding decay rates for the purple curves imply a very fast cooling schedule regardless of which form it takes due to local optimum convergence.

**Parameter Tuning for Genetic Algorithm**

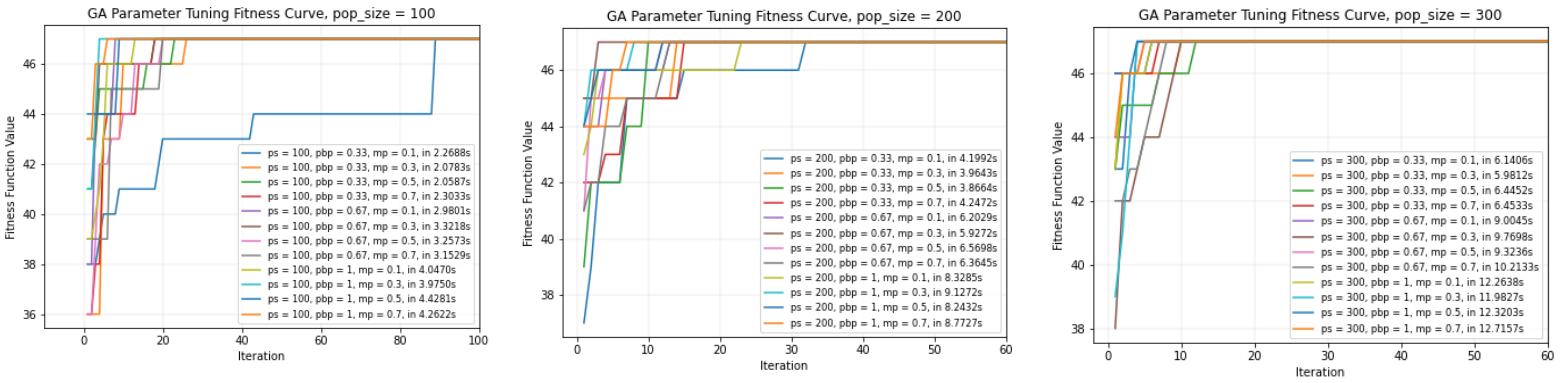



Figure 14

**Selected Parameters**

| Algorithm | Max Attempts | Restarts | Average Runtime |
|---|---|---|---|
| RHC | 500 | **800** | **6.56** |

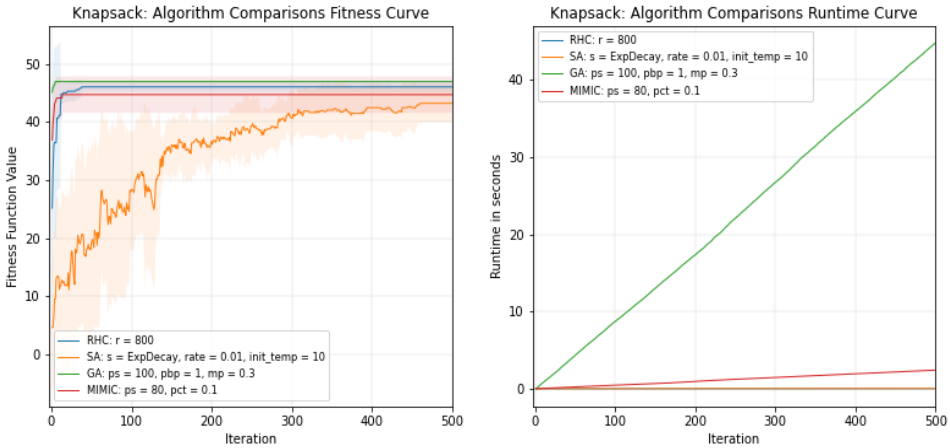
| Algorithm | max-attempts | Init Temp | Form | Decay Rate | Average Runtime |
|---|---|---|---|---|---|
| Simulated Annealing | 8000 | 1, 5, **10** | Arithmetic, **Exponential**, Geometric | Arithmetic = [0.0001,0.001,0.01], Exponential = [0.005, **0.01** , 0.02], Geometric = [0.999, 0.99, 0.9]] | 0.1410s |

| Algorithm | max_attempts | Pop_size | pop_breed_percent | Mutation_prob | Average Runtime |
|---|---|---|---|---|---|
| GA | 500 | **100** | 0.33, 0.67, **1** | **0.3** | 3.9750s |

| Algorithm | max_attempts | pop_size | pct_kept | Average Runtime |
|---|---|---|---|---|
| MIMIC | 500 | 20,40,**80** | **0.1**,0.3, 0.5, 0.7 | 2.7013s |

**Algorithms Comparison: Mean Fitness Curve and Computation Times**

Strangely the pop_breed_percent for genetic algorithm is shown to be best performing at 1 which means to breed all of the population. MIMIC turns out to be the least performing in terms of the time and iteration it takes to converge, infinity in this case. The genetic algorithm takes the longest time, about 0.9 seconds per iteration but it has attained the global optimum in the first 20 iterations. MIMIC and RHC failed to converge at the global optimum.




| Algorithm | Runtime when max_attempt = 1000 |
|---|---|
| RHC | (No Convergence) |
| SA | 13.64299 |
| GA | 9.12149 |
| MIMIC | (No Convergence) |

Convergence occurred relatively quickly compared to other problems so xlim for the plots have been modified for clear comparison. The winner in this case is genetic algorithm due to its ability to reach the global optimum in the early iterations for five different seeds. Although the amount of time it takes is the largest per iteration the amount of iterations can be cut off early.

Figure 14

5. Supervised Learning Problem: The Mushroom Dataset

The mushroom.csv from UCI ML dataset repository (https://archive.ics.uci.edu/ml/datasets/mushroom) has 22 categorical variables of physical traits of the 8,124 observations of mushroom observations that are within 23 species of gilled mushrooms in the Agaricus and Lepiota Family using descriptive encoding and 1 output variable that indicates whether the observation is poisonous or edible. The train/test set is split 4:1.

These 22 categorical variables are then turned into 95 binary dummy variables effectively making this a 95 bit string state space. A neural network is built using GridSearch to find the best performing hidden layer nodes and activation function in terms of 5-fold CV accuracy. Its weights are then calculated based on four optimization methods: backpropagation (gradient descent), random hill climb, simulated annealing and genetic algorithms.

5.1 Optimization of Neural Net Weights – Architecture Parameter vs Optimizer Parameter

To ensure the fair comparison of the various randomized optimization algorithms and the backpropagation via gradient descent, the neural network's hidden layer architecture (i.e., width, depth, total number of nodes, etc) and activation function must remain consistent. The hidden layer architecture is determined using a GridSearch of the highest 5-fold cross-validation accuracy on the training set of the mushroom dataset with gradient descent via sigmoid activation function by testing architecture candidates that are purported to perform well in supervised algorithms with respect to the number of training samples, input and output layer. The following link has provided insight into to picking out hidden_nodes array candidates to grid search for: https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

The grid searched number of hidden_nodes are as follows: [25], [20], [10], [25,10], [25,5], [20,10], [10,5], [20,5]. And the GridSearch returned [25].

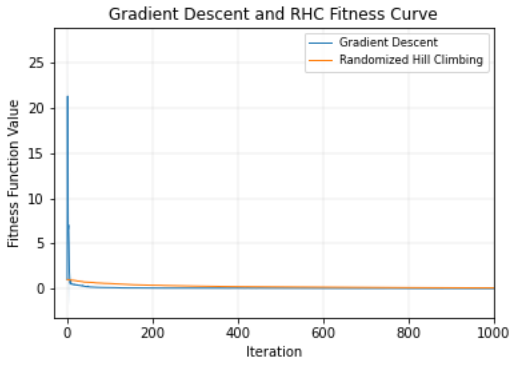
The parameters so far define the optimization problem which the methods aim to solve. Other parameters of the MLROSe neural network instance such as the learning rate and max attempts along with the optimization method's native hyperparameters (e.g. "restarts" for Randomized Hill Climbing and "decay schedule" for Simulated annealing) define the optimization method. These other parameters are also 5-fold CV grid searched but separately for each optimization method and outputting the best set of parameters per the four methods: gradient descent, randomized hill climb, simulated annealing and the genetic algorithm. Each randomized optimization algorithm is run 10 times to calculate the mean train/test accuracy to confirm the CV results.

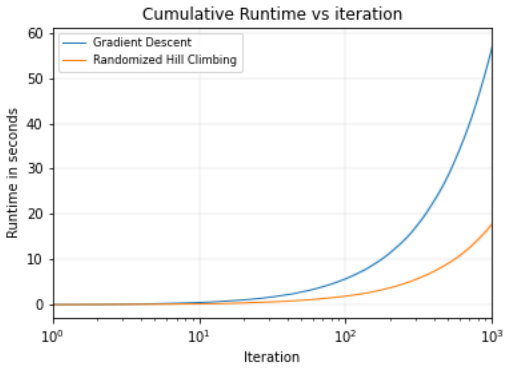
5.2 Optimization of Neural Net Weights – Loss Function

The MLROSe-hiive for the Neural Network instance when probed into the library shows sklearn.metrics.log_loss is what is used in terms of this classification not the accuracy rate. The appended fitness value each iteration for the fitness curve is from this log-loss function evaluation: -log P(yt|yp) = -(yt log(yp) + (1 - yt) log(1 - yp)) and it is something to minimize (despite the conventional notion that fitness function should be maximized unlike cost functions), where yt is the true value (0 or 1) and yp is the predicted value. The global optimum is 0.

5.3 Randomized Hill Climbing Parameter Tuning

The learning rate of the randomized hill climb is varied in this manner below: [0.01, 0.1, 1, 1, 5, 10, 20]. It shows faster and optimal convergence at 20. As the learning rate rises, it is shown from all randomized optimization algorithm that the Test Accuracy steadily increases. Above figure shows that when the gradient descent learning rate is changed from 0.1 to 10 it produces spikes at certain iteration likely due to too much incorrect changes. The "restarts" parameter seems to multiply the runtime by the number specified. It turns out restarts = 30 is enough retries for exploration. Any more or less of it does not result in a higher test accuracy or faster convergence in iteration and results in more runtime.




Gradient Descent: mean train accuracy 0.979209

Gradient Descent: mean test accuracy 0.985240

Randomized Hill Climbing: mean train accuracy 0.987260

Randomized Hill Climbing: mean test accuracy 0.988403

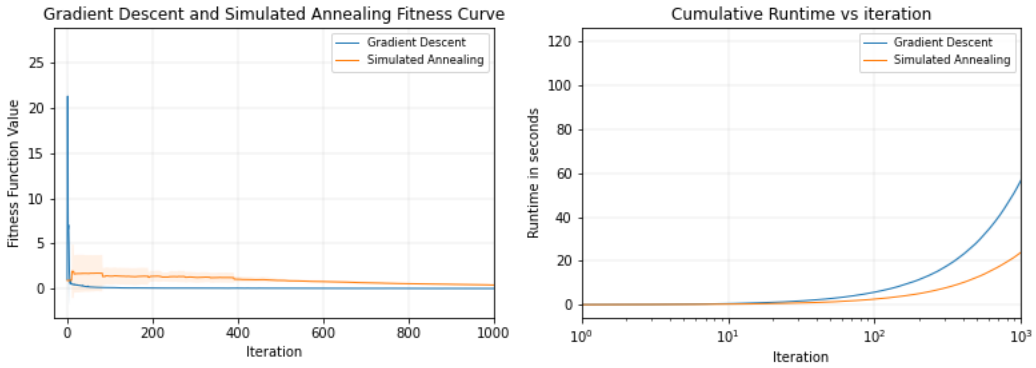
Figure 15

The max_iters is set as 1000 and the max_attempts is set as 200. The randomized hill climb consumes less time relative to gradient descent and does a decent job of converging within 1000 iterations with a mean test accuracy of 0.99. It starts off not too high suggesting that the method approaches

| RHC Learning Rate | 0.01 | 0.1 | 1 | 5 | 10 | 20 | 40 |
|---|---|---|---|---|---|---|---|
| Test Accuracy | 0.493226 | 0.51108 | 0.954433 | 0.969827 | 0.985221 | 0.991752 | 0.991379 |

5.4 Simulated Annealing Parameter Tuning

The max_attempts is set to 200 though it turns out SA constantly has changes in the loss curve due to poor fitness acceptance. The max_iters = 1000 has been shown to be insufficient to converge as well as gradient descent does since it exhibits a steady downward curve with the potential to converge more toward the global minimum. The GridSearchCV returns Geometric decay using rate = 0.99 as the cooling schedule based on three tries. Upon further experimentation using a rate = 0.9 is shown to converge faster with a steeper slope. However, when run with different seeds the test accuracy occasionally does not improve after a certain point. This suggests that some hotness (exploration) is welcomed to avoid converging to a local minimum. The arithmetic approach has been shown to be very slow to converge and results in a very spiky loss curve in the early iterations even up to 1000. Similar plots are observed with init_temp = 10 due to slower cooling.




Gradient Descent: mean train accuracy 0.979209

Gradient Descent: mean test accuracy 0.985240

Simulated Annealing: mean train accuracy 0.971604

Simulated Annealing: mean test accuracy 0.975381

Figure 16

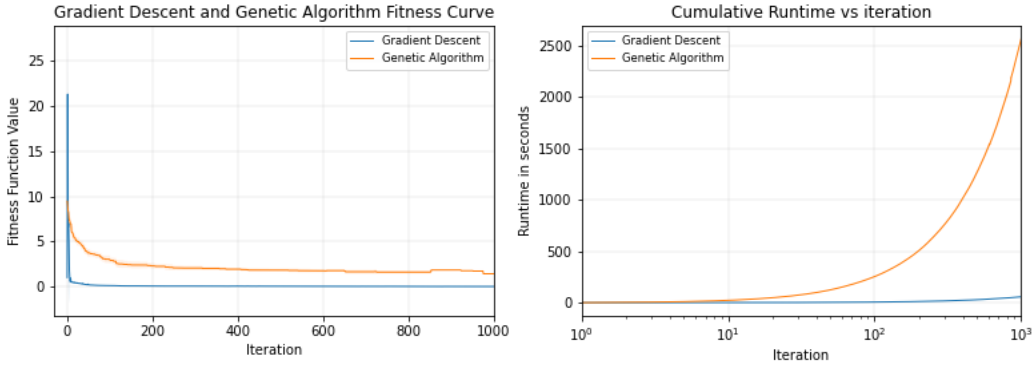
| SA Learning Rate | 0.1 | 1 | 10 | 20 | 40 |
|---|---|---|---|---|---|
| Test Accuracy | 0.769941 | 0.865953 | 0.971689 | 0.975381 | 0.975381 |

*The learning rate has converged in test accuracy for 20+ in one experiment, and it is possibly indicative of becoming stuck at a local minimum.

The simulated algorithm exhibits a band of deviation that narrows as the iteration increases. However, it is shown that the algorithm showed convergence in much later iterations, showing a steady convergence but still fast in terms of wall clock time which requires the max_iters to be set a few times higher to converge than is required for other methods. The decay rate for the temperature must be chosen carefully so it neither over-explores nor over-exploit.

5.5 Genetic Algorithm Parameter Tuning

The parameters pop_size has initially been set to 100 and experienced severe runtime so it has been adjusted to 30. The initial run which has been stopped has been revealed to take two hours to reach the first 1000$^{th}$ iteration. It has not performed nearly as well as backpropagation has via gradient descent within the 1000$^{th}$ iteration. It shows steady convergence that is too slow in runtime and in iterations. Then on the next run with one seed it has been revealed that setting pop_size to 30 resulted in an even poorer performance. Mutation probability has been set to 0.05 as it showed higher mean test accuracy for runs with max_iters = 1000 than mutation_prob = 0.1, which suggests that it has explored excessively.




Gradient Descent: mean train accuracy 0.979209

Gradient Descent: mean test accuracy 0.985240

Genetic Algorithm: mean train accuracy 0.953221

Genetic Algorithm: mean test accuracy 0.954490

Figure 17

The GA starts off with a high test accuracy, or equivalently, a low loss function value and requires a relatively low learning rate of 0.01 compared to RHC and SA. It is shown that GA has expired early at a local optimum at around 600$^{th}$ iteration due to max_attempts limit of 200. The Genetic Algorithm takes long to converge or expire in terms of iteration and wall clock time. It is highly inefficient. The pop_size has a direct relationship to the runtime of the algorithm. A high value of mutation_prob, unlike in the other simple optimization problems, does not necessarily result in a better performance in test accuracy as well as reducing computation time. Too much in fact resulted in poorer fitness function value likely due to excessive exploration.

| GA Learning Rate | 0.01 | 1 | 10 | 20 |
|---|---|---|---|---|
| Test Accuracy | 0.955720 | 0.954490 | 0.955720 | 0.955720 |

6. All Four Weighting Algorithms Compared

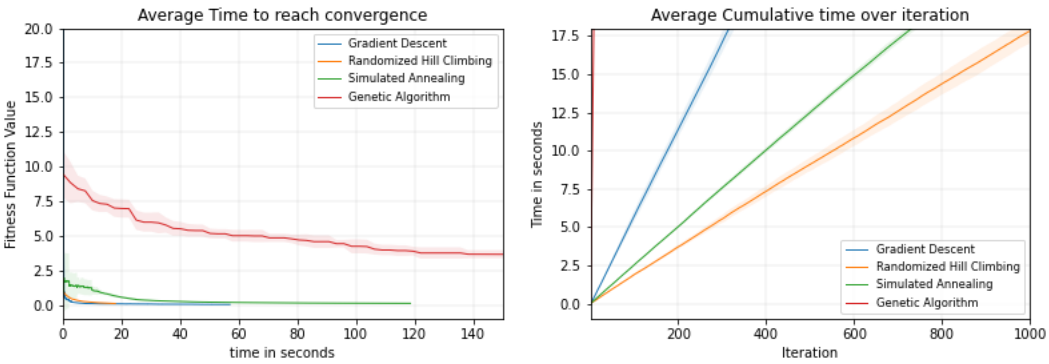



Figure 19

Backpropagation via gradient descent of learning rate = 0.01 is the best performing optimization method due to the convergence speed relative to the number of iteration and wall clock time while producing high enough mean test accuracy 98.5240%. It explores heavily in the first 10 iterations and converges rapidly within the 100th iteration all the time.

- The randomized hill climbing takes the least runtime per iteration and takes the 2nd least iterations to converge (or expire due to no improvement), and this runtime is observed to scaled linearly by the number of restarts specified.
- The simulated annealing with a geometric decay with rate = 0.99 takes a lot of iterations to converge despite having relatively low runtime per iteration. It explores more than other algorithms in the first 100th iteration but it steadily and quickly (in runtime) converges to the global optimum flaunting a test accuracy of 0.975381. Increasing the init_temp from 5 to 10 is shown to result in a slower overall convergence (186.1548 seconds).
- Genetic Algorithm is highly inefficient due to very high wall clock time. At this point increasing the population size, mutation_probability or the elite-to-dreg ratio has been fruitless since the amount of accuracy and faster convergence achieved from it does not outweigh the cost in terms of the additional time it takes to run one iteration. Requires low learning rate compared to Randomized Hill Climbing and Simulated Annealing (0.01 vs 10). The loss function decreases slowly and requires many heavy iterations to converge.
- Aside from the perspective of optimization, this learning task seems easy to the Neural Network's architecture regardless of how weights are optimized in the sense that the test accuracy is excellent throughout the multiple seed with various reruns even with shuffled train/test set split.
- In the case of weighting neural nets in this problem, too much exploration can hurt since there are certain key dummy variables, or bits, out of the 95 that are the key activators or features of the poisonous classification, i.e. has very high feature importance.

7. Conclusion

Like the bias-variance trade-off observed in supervised learning models, the optimization algorithms have a trade-off between their exploration and exploitation. The parameters must be tuned exactly according to the problem it is trying to solve to neither over-explore nor over-exploit. Also, some algorithms are better equipped to solve certain problems, while others are not. The randomized algorithms tuning process requires various perspectives such as wall clock time, number of iterations needed, memory needed and its ability to converge at all to the global optimum of the problem. Even if infinite computational time and space are provided, a randomized optimization algorithm can get stuck in a local optimum after some iteration for certain problems. Finally, in the case of neural network weights, no optimization methods outperform backpropagation given the same iterations/runtime limit.