# Randomized Optimization

Elijah Rockers
CS-7641

## Abstract

This submission will explore four different randomized optimization (RO) machine learning methods, as applied to several bit string discrete optimization problems where the algorithm determines at every iteration which "direction" to move the bit string in to increase fitness. Each algorithm implements a "randomness" factor to help avoid local optima, with the ultimate goal of finding a global maximum.

These algorithms are: Randomized Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithms (GA), and Mutual Information Maximizing Input Clustering (MIMIC).

In addition, we will see how three of the four methods (RHC, SA, GA) can be applied to find the optimal weights of the neural network described in Assignment 1, against the Beans dataset, which is a multiclass classification problem.

The three RO problems include:

**Four Peaks**: This problem models an N-dimensional (N=40 in this case) vector X that includes two sharp global maximum on the edges, and two suboptimal local maxima that occur on a plateau with a reward function defined as

$$F(X, T) = \max[Tail(0, X), head(1, X)] + R(X, T)$$

Tail(0, X) = number of trailing 0's in X

Head(1, X) = number of leading 1's in X

And the reward R(X, T) is N depending on the T threshold value, and 0 otherwise.

On the plateau, it's likely that hill climbing or simulated annealing algorithms would converge to a local maxima, while not finding the global maximums. This problem is designed as a kind of "trap" for these algorithms, and it's expected that the genetic algorithm might be able to find the global maxima, as it doesn't depend on a typical geometric representation of fitness (e.g. gradient descent). In the N=40 case, the maximum fitness was F = 75.0.

**N-Queens**: This problem models an NxN (N=20 in this case) chess board, where the optimization problem attempts to find a queen position for each rank (or file) where no Queen is not attacking any other. This problem can be represented as an N-vector, with maximum value N-1, each element representing a rank, and each value of the element representing the file to place the Queen on. These problems can be solved by algorithmic backtracking, but this particular assignment focuses on solving the problem with randomized optimization. For the N-Queens problem, fitness was maximized at F = 24.0.

**Knapsack**: The knapsack problem is a classic example of a combinatorial optimization problem, where the goal is to find the best combination of items to put in a knapsack (or backpack) with limited capacity, such that the value of the selected items is maximized while not exceeding the weight capacity of the knapsack.

Given a set of N items, where each item has a value and a weight, and a knapsack with capacity W, the goal is to select a subset of the items such that the total weight of the selected items does not exceed the capacity of the knapsack, i.e., $\sum$weights $\leq$ W, and the total value of the selected items is maximized, i.e., $\sum$value is maximized.

For this Knapsack problem, the weights and corresponding values are displayed below, representing a maximum possible fitness of F = 61.0:

**Weights**: [10, 5, 2, 8, 15, 11, 4, 7, 1, 20, 4, 60, 30, 55]

**Values**: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

For each of these problems, we tested all four RO algorithms with varying hyperparameters, as described below:

**SA**:

Iterations = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
Temperature = [0.1, 0.5, 0.75, 1.0, 2.0, 5.0] with Geometric decay.

**GA**:

Iterations = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
Population Size = [150, 200, 300]
Mutation Rate = [0.4, 0.5, 0.6]

**MIMIC**:

Iterations = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
Population Size = [150, 200, 300]
Keep Percentage = [0.25, 0.5, 0.75]

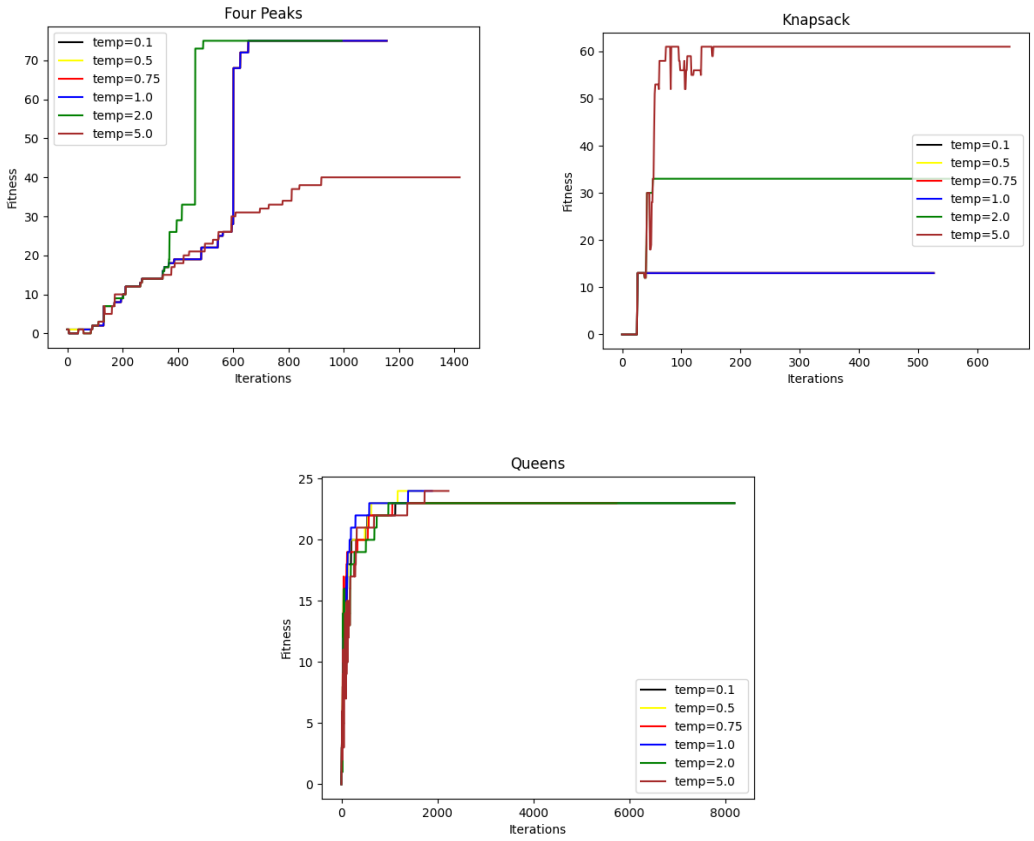
**RHC**:

Iterations = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
No restarts.

# Results

## Simulated Annealing

Simulated annealing is used to find the global minimum (or maximum) of a complex, multi-dimensional function that may have many local optima. It does so by searching the solution space randomly, accepting worse solutions with some probability in order to escape local optima, and gradually decreasing the probability of accepting worse solutions as the search progresses. As iterations progress, the temperature decreases, increasing the likelihood of converging to the optimal solution. In the case of discrete bit string optimization, neighbors are evaluated with a distance of 1 bit difference from the current string.

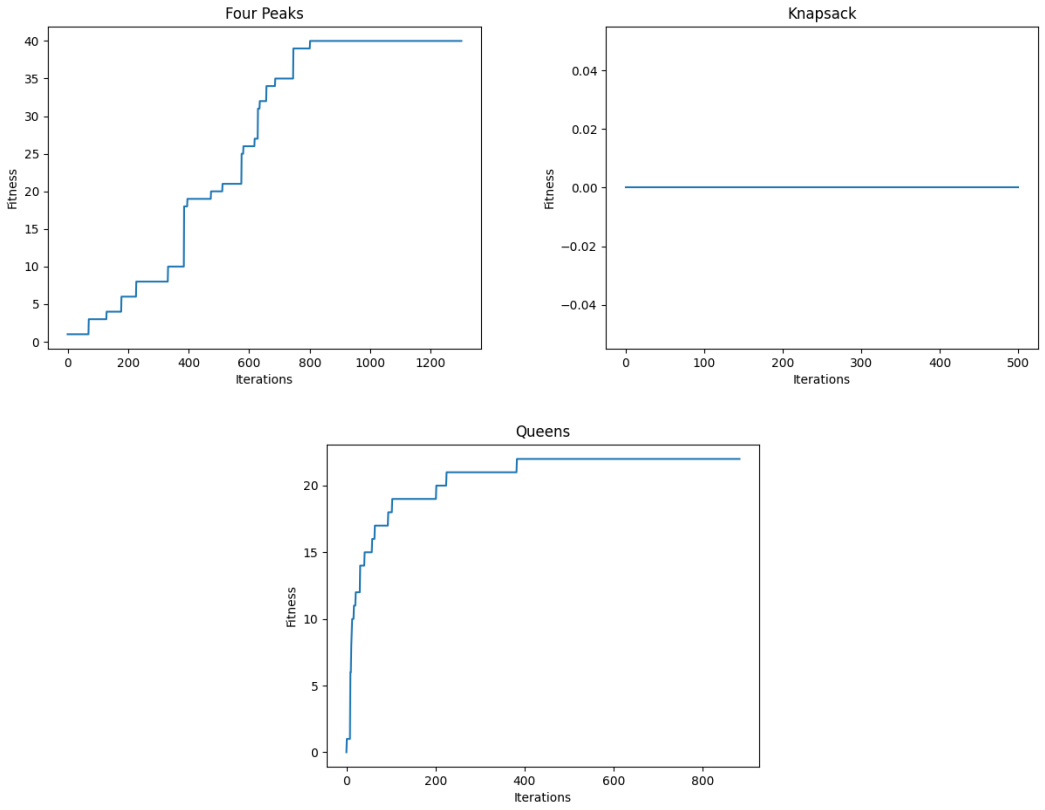



For the three problems, simulated annealing seemed to work best (maximizing fitness in the least amount of iterations) when solving the 20-Queens problem. While the other two problems did eventually converge, convergence was highly dependent on the temperature value.

## Randomized Hill Climbing

The main idea behind RHC is to start with an initial solution and iteratively improve it by making small, random modifications to the solution and keeping the new solution if it is better than the current solution. RHC is simple and efficient. The random modifications made to the current solution are

typically small changes, such as flipping a bit in a binary string, or adding or subtracting a small value in a continuous function. These modifications are made randomly to avoid getting stuck in local optima.

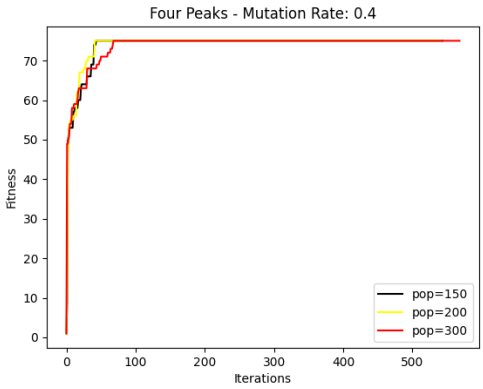


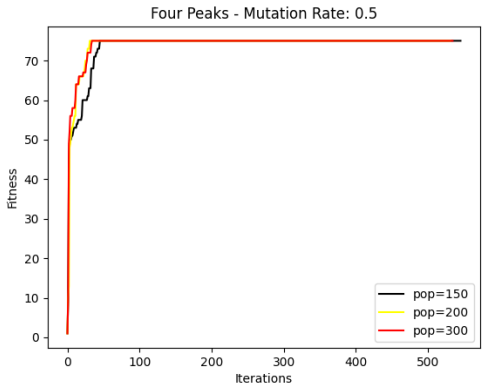

For the three problems, randomized hill climbing seemed to work best on the N-Queens problem reaching a solution in less iterations than on the Four Peaks problem. Interestingly, it failed completely on the Knapsack problem. This could be due to the initial conditions of the bit string, and the completely random nature of RHC. Four Peaks requires a fairly specific configuration of the 40 dimensional bit string which is unlikely to be achieved with RHC in a reasonable amount of iterations, and is likely to converge to a local optima. RHC perhaps most closely resembles what we might consider to be a backtracking algorithm, which may explain why it works well on the 20-Queens problem.

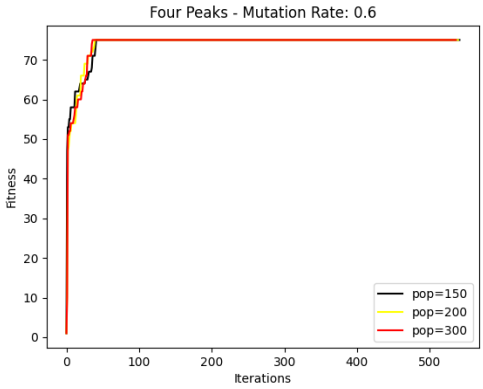
## Genetic Algorithm

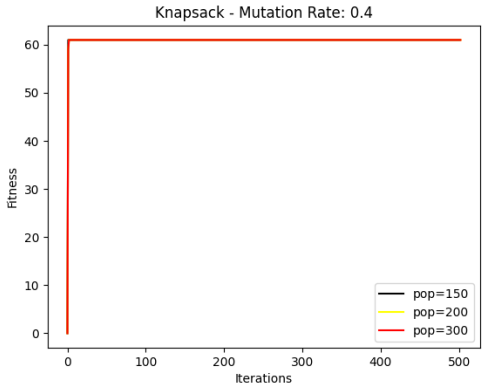
GA is a randomized optimization algorithm that is inspired by the process of natural selection and evolution. It works by iteratively generating a population of candidate solutions to a problem, evaluating their fitness, and selecting the best solutions for reproduction and mutation to create the next generation of solutions. The algorithm is based on the idea that good solutions can be created by combining and mutating existing solutions, in the same way that nature creates new species through genetic recombination and mutation. Initial conditions include the population size, which is typically

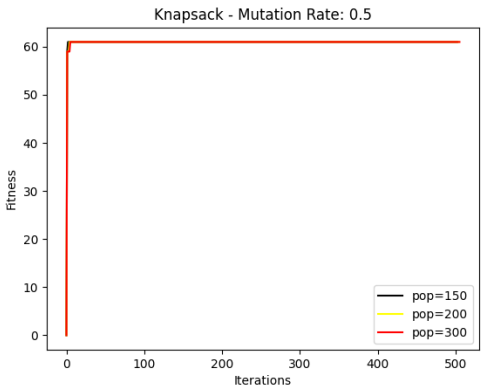
determined by trial and error, and depends on the dataset. A large population size can lead to long running times. The mutation rate determines the probability of a bit mutation during each generation.

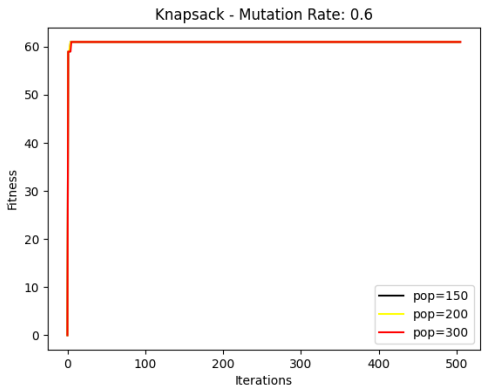


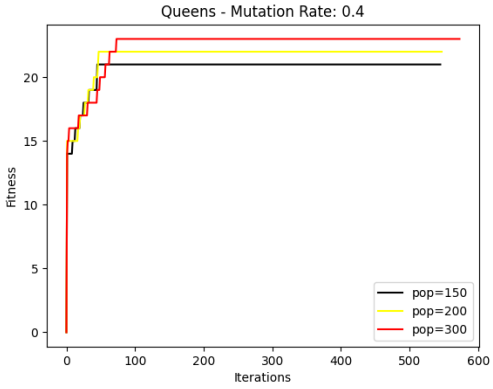



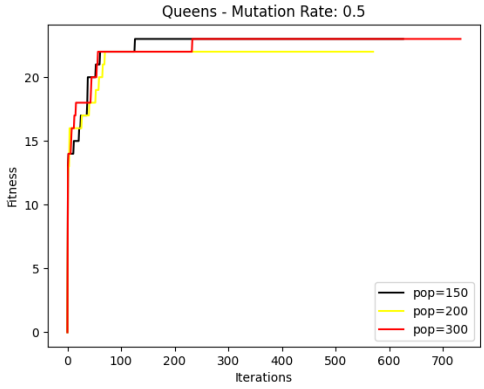



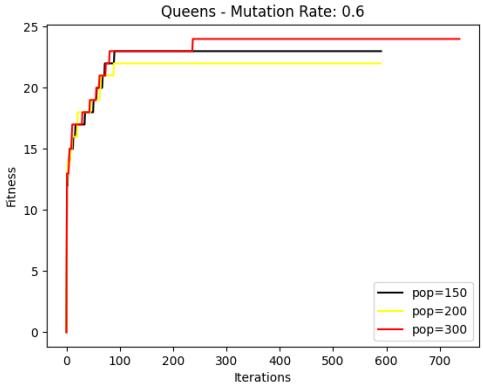





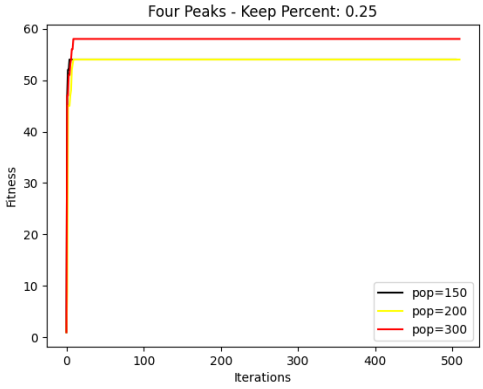



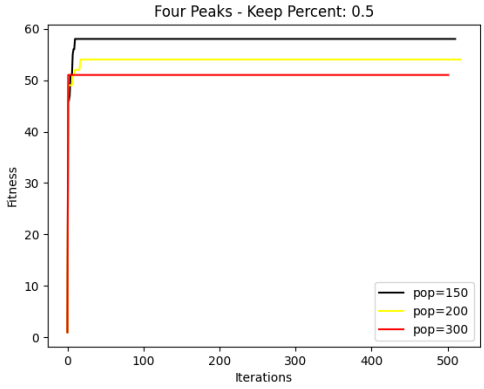



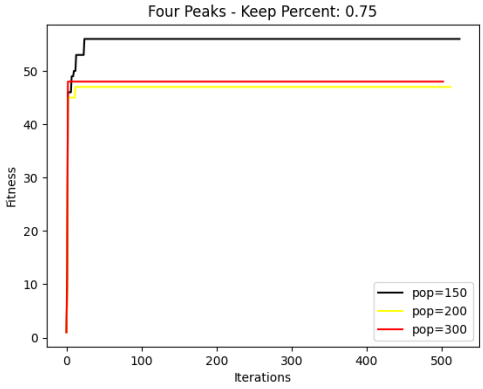

The genetic algorithm worked fairly well on all problems, but perhaps least efficient on the 20-Queens problem. Population size and mutation rate seemed to have very little effect given the problems proposed. The genetic algorithm converged extremely quickly for the knapsack problem, and as expected, for the Four Peaks problem. For the 20-Queens problem, it came close, but only converged to the true maximum with a mutation rate of 0.6 and population of 300.

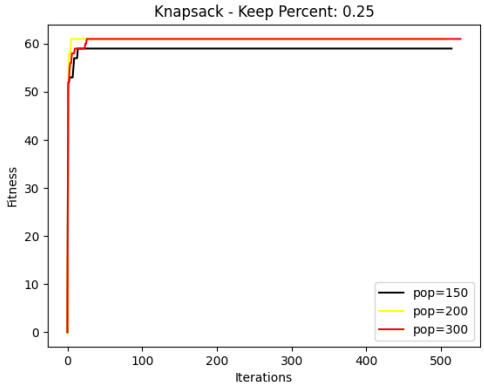
## MIMIC

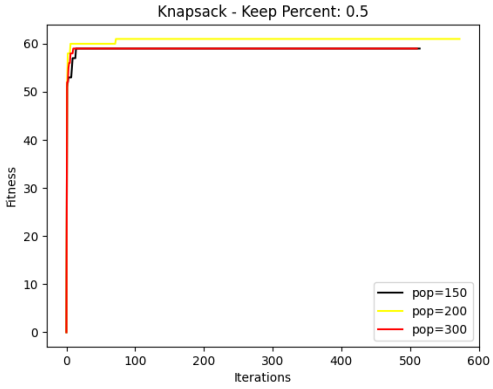
The main idea behind MIMIC is to use a probabilistic model to learn the structure of promising solutions in the search space, and then use this model to generate new candidate solutions that are likely to be good. MIMIC uses a joint probability distribution over the variables in the problem. This distribution is learned by estimating the mutual information between pairs of variables in the population, which captures the degree to which the variables are dependent on each other. The learned distribution is then used to generate new candidate solutions by sampling from the distribution. The keep percentage refers to the proportion of the best solutions from the current population that are selected to be used for further optimization. The remaining solutions are discarded, and new solutions are generated by sampling from the learned probabilistic model.

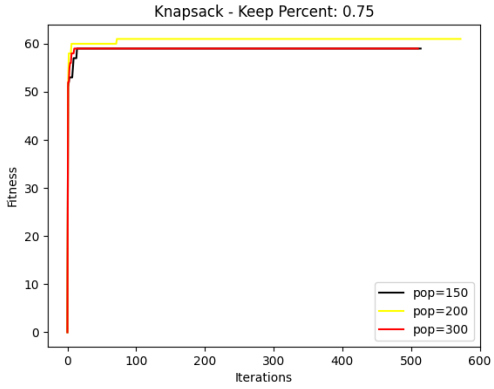

Four Peaks - Keep Percent: 0.25
Fitness
Iterations
pop=150
pop=200
pop=300

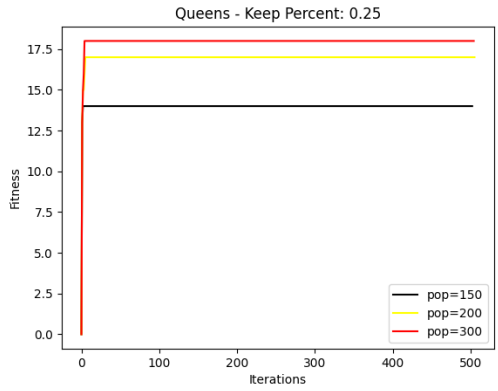

Four Peaks - Keep Percent: 0.5
Fitness
Iterations
pop=150
pop=200
pop=300

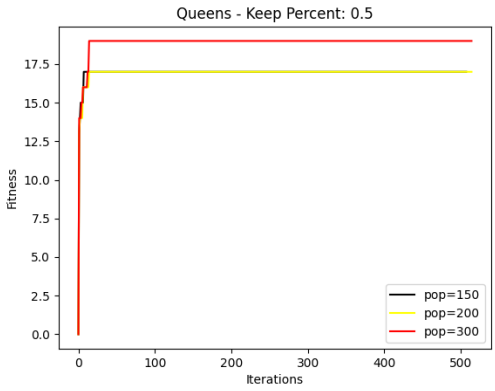

Four Peaks - Keep Percent: 0.75
Fitness
Iterations
pop=150
pop=200
pop=300

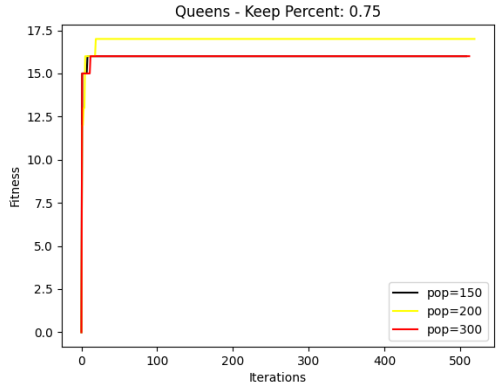

Knapsack - Keep Percent: 0.25
Fitness
Iterations
pop=150
pop=200
pop=300


Knapsack - Keep Percent: 0.5
Fitness
Iterations
pop=150
pop=200
pop=300


Knapsack - Keep Percent: 0.75
Fitness
Iterations
pop=150
pop=200
pop=300




The MIMIC algorithm converged to the global optima for the knapsack problem with a keep percentage of 25%, and larger populations, and came close when applied to the four peaks problem.

## Neural Network Randomized Optimization

The first three (SA, RHC, and GA) algorithms were then used to attempt to optimize the weight of a neural network with the architecture described in Homework 1 [500, 500, 10]. Randomized optimization can be used to train neural networks by searching for the optimal values of the network's parameters that minimize a given loss function, which in this case is minimizing log loss. Randomized optimization algorithms can be effective for training neural networks, especially when the search space is large or gradient descent is less effective. However, randomized optimization algorithms can be computationally expensive, and it can be difficult to know when the algorithm has found the global minimum of the loss function.

### *Simulated Annealing*

For SA, learning rates of [0.1, 1, 10, 100] were explored, with two activation functions ReLU and tanh. Two different decay schedules with initial temperatures [1, 10] were also attempted with a decay rate of 0.0001.

Log loss was minimized with a learning rate of 0.1, activation ReLU, and an initial temperature of 1.

### *Genetic Algorithm*

For GA, learning rates of [0.1, 1, 10, 100] were explored, with two activation functions ReLU and tanh. Population sizes of [10, 20, 30] were tried as well as mutation rates of [0.2, 0.4] using a grid search.

Log loss was minimized with a learning rate of 0.1, activation ReLU, population size of 20, and mutation rate of 0.2.

### *Randomized Hill Climbing*

For RHC, learning rates of [0.1, 1, 10, 100] were explored, with two activation functions ReLU and tanh.

Log loss was minimized with a learning rate of 0.1, and activation ReLU.