Randomized Optimization

Travis Latzke

October 13, 2019

1 Introduction

Optimization problems generally try to maximize a value $f(x)$ where $f : S \rightarrow \mathbb{R}$ is an evaluation function and S is the set of all possible solutions. The value x $\in S$, such that $f(x) > f(x') \; \forall x' \in S$, is considered to be the global optimal solution and the value $f(x)$ is considered to be the global optimal value. Finding the optimal solution is usually considered to be more difficult when f contains many local maximum points because search algorithms tend to get stuck at local optimal values. For example, greedy algorithms tend to get "stuck" at local maximum points because the algorithm is instructed to move to a new point only if the current point has a neighbor whose evaluation score is higher than the current point. Therefore, the point that is initially chosen as the starting point for greedy algorithms is a vital factor in determining whether the algorithm finds the global optimal solution. However, some problems have a large solution space, so selecting a good starting point is almost as difficult as finding the solution itself. Thankfully, there are a class of algorithms that use a technique called randomized optimization to prevent algorithms from getting stuck at local maximum solutions. The algorithms explored in this paper are randomized hill climbing (RHC), simulated annealing (SA), genetic algorithms (GA), and MIMIC. Each algorithm uses some form of randomization that is controlled by hyper-parameters that are specific to each algorithm. The hyper-parameters used in each algorithm are explained below.

Maximum Iterations: *The number of times the algorithm searches for a neighbor that produces a higher optimal value than the current best solution* (RHC, SA, GA, MIMIC).

Maximum Attempts: *The maximum number of consecutive searches that result in the neighbor having a lower optimal value than the current best solution* (RHC, SA, GA, MIMIC).

Temperature T: *The temperature is proportional to the probability in which the algorithm moves to a random neighbor that is less optimal to the current point* (SA).

Temperature Decay Rate T_γ: *The rate to decay the temperature as a function of the number of iterations* (SA).

Population P: *The rate to decay the temperature as a function of the number* (GA, MIMIC).

Population to Mate P_M: *The proportion of the population that has a high enough fitness to breed the next generation* (GA).

Mutation Rate M_γ: *The probability of a mutation to occur during the crossover phase of two parents* (GA).

To Keep k: *The number of samples to keep from the prior iteration* (MIMIC).

1.1 Traveling Salesman Problem

The traveling salesman problem involves finding the shortest distance that needs to be traveled in order to visit n cities exactly once in addition to ending the trip in the same city that the trip started in. More formally, the goal is minimize $h(x)$:

$$x_{ij} = \begin{cases} 1 & \text{if a path exists from city i to city j} \\ 0 & \text{otherwise} \end{cases}$$

$$h(x) = \sum_{i}^{n} \sum_{j}^{n} d_{ij} x_{ij}$$

Where d_{ij} is the distance from the city i to city j. The size of solution space for this problem is $n!$ so it is impractical to use a brute force algorithm to solve this problem. Fortunately, randomized optimization has proven to be a successful technique to approximating a good enough solution for problems involving large values of n. The evaluation function for this problem is the inverse of $h(x)$ to account for each algorithm trying to maximize $f(x)$.

$$f(x) = 1/h(x)$$

Figure 1 shows how each algorithm performed in searching for an approximate solution to the traveling salesman problem. Each algorithm was run multiple times with a different set of hyper-parameters.

SA	T	T_γ
model 1	1e12	0.99
model 2	1e12	0.9
model 3	1e12	0.8
model 4	1e12	0.7
model 5	-	-

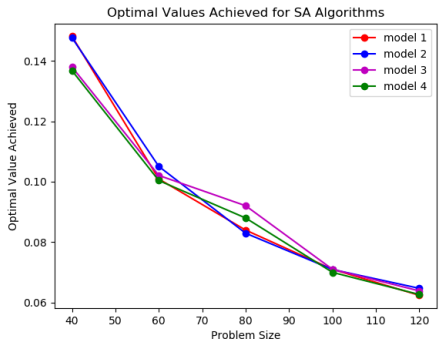
GA	P	P_M	M_γ
model 1	200	0.75	0.2
model 2	300	0.66	0.25
model 3	400	0.625	028
model 4	500	0.6	0.3
model 5	600	0.58	0.314

MIMIC	P	k
mode 1	150	80
mode 2	200	100
mode 3	250	120
mode 4	300	160
mode 5	350	200

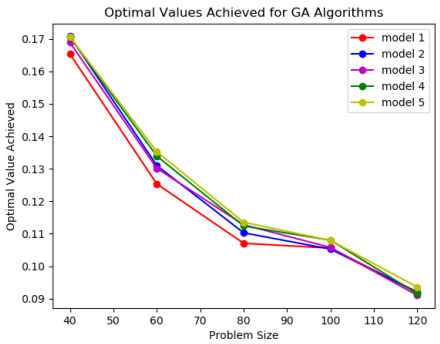
The results show that the genetic algorithm consistently outperformed all of the other algorithms in finding the best optimal value for each traveling salesman problem. This could be attributed to the way the algorithm generates new children each iteration. Children are generated by using the 'salesman' crossover method. This could be an effective method for breeding because parent A might have found a good solution for the cities $1 - m$, and parent B could have a good solution for the cities $m - n$. If the parents make a child at the m^{th} crossover point, then the child could have the good solutions from both parents. Being that the crossover point is random, a higher population ensures a higher probability of generating a child that is the best version of both parents. However, raising the population comes at a cost by increasing the number of function evaluations done by the algorithm. This explains the large number of function evaluations done by the genetic algorithm in comparison to simulated annealing and RHC. SA and RHC most likely got stuck at local optimal values, which explains the reason for the algorithms achieving a low optimal value with a small number of function evaluations in comparison to the genetic algorithm.

Algorithm	$n = 40$	$n = 60$	$n = 80$	$n = 100$	$n = 120$
RHC	4495	14791	29318	72728	430678
SA	199965	199885	199908	199699	199837
GA	430818	548314	548264	430678	548217
MIMIC	500231	700480	600436	700394	1395359

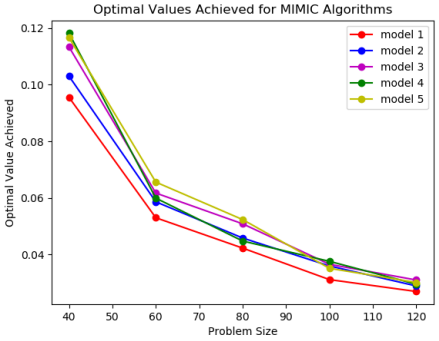
Table 1: *The number of function evaluations calculated for each algorithm to achieve it's corresponding optimal value for the traveling salesman problem*



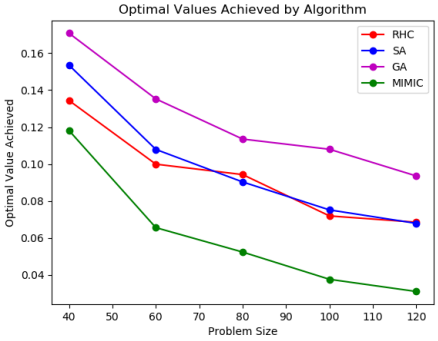
(a) Simulated Annealing



(b) Genetic



(c) MIMIC



(d) Best of each

Figure 1: *Comparison of optimal values achieved by different optimization algorithms on the traveling salesman problem*

1.2 Flip Flop Problem

The flip flop problem can easily be defined using bit strings with variable sizes of length n. The goal of the problem is to maximize the number of alternating occurrences of 1's and 0's in a n bit string:

$$h(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

$$f(x) = \sum_{i=0}^{n-1} h(x_i, x_{i+1})$$

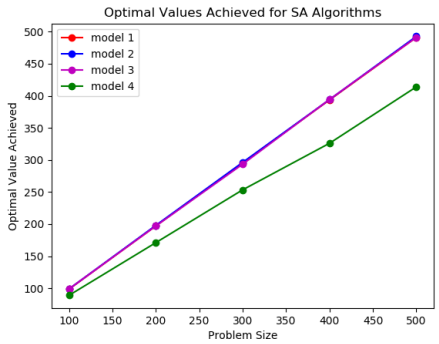
Similar to the traveling salesman problem, the optimization algorithms were run with different hyper-parameters to measure the performance of each algorithm in comparison to one another.

SA	T	T_γ
model 1	1E5	0.99
model 2	1E5	0.8
model 3	1E5	0.6
model 4	1E5	0.4
model 5	-	-

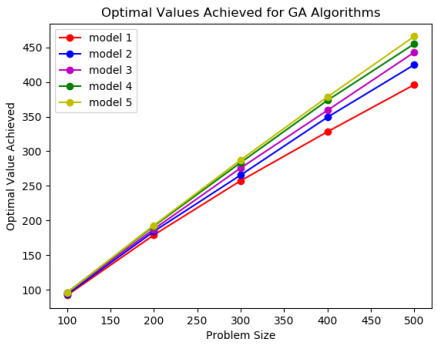
GA	P	P_M	M_γ
model 1	500	0.6	0.16
model 2	1000	0.6	0.16
model 3	2000	0.6	0.16
model 4	5000	0.6	0.16
model 5	10000	0.6	0.16

MIMIC	P_{size}	k
mode 1	100	5
mode 2	200	10
mode 3	300	15
mode 4	400	20
mode 5	500	25

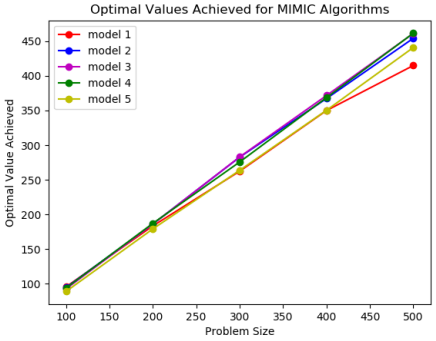
The genetic algorithm performed much worse on the flip flop problem in comparison to how it performed on the traveling salesman problem. The GA results were gathered using the single crossover method, but it is likely that other crossover methods would perform similarly. The reason being due to the difficulty in two parents generating a child with a higher fitness. For example, assume parent A has a higher fitness than parent B because parent A has alternating bit values from the 0^{th} index



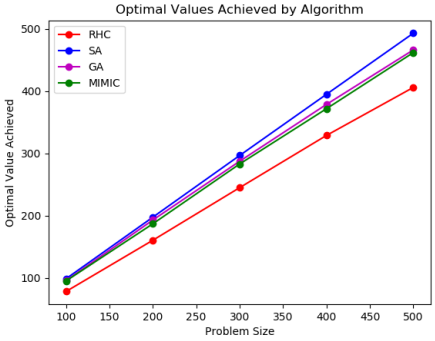
(a) Simulated Annealing



(b) Genetic



(c) MIMIC



(d) Best of each

Figure 2: *Comparison of optimal values achieved by different algorithms on the flip flop problem*

Algorithm	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$
RHC	598	1252	2079	3607	4850
SA	199965	199885	199908	199699	199787
GA	6380569	12761664	12761411	12761378	12761294
MIMIC	200101	666982	600303	400201	400138

Table 2: *The number of function evaluations calculated for each algorithm to achieve it's corresponding optimal value for the traveling salesman problem*

to the m^{th} index of an n-bit string. Assume parent B has some random arrangement of bits. If c is the crossover point and is chosen randomly on the interval $[0 - n]$, and if $c < m$, then the child's fitness will likely be reduced by a factor of $m - c$ from parent A's fitness. If $c > m$ then the child will have a fitness similar to parent A's fitness because parent A and B both contain random bits after the m^{th} position of their bit strings. In this case, the algorithm breeds a higher portion of children with lower fitness levels than their parents. That suggests that the algorithm should only breed a small fraction of the population with the best genes, so that unfit children are weeded out. This allows the algorithm to continue to improve on finding an optimal value, but at a very high cost in terms of time and the number of function evaluations. The Simulated Annealing algorithm runs much quicker and achieves a better optimal value because the core hill climbing algorithm in SA allows the algorithm to improve as fast as it can select a random neighbor with more alternating bits.

1.3 Max K Coloring Problem

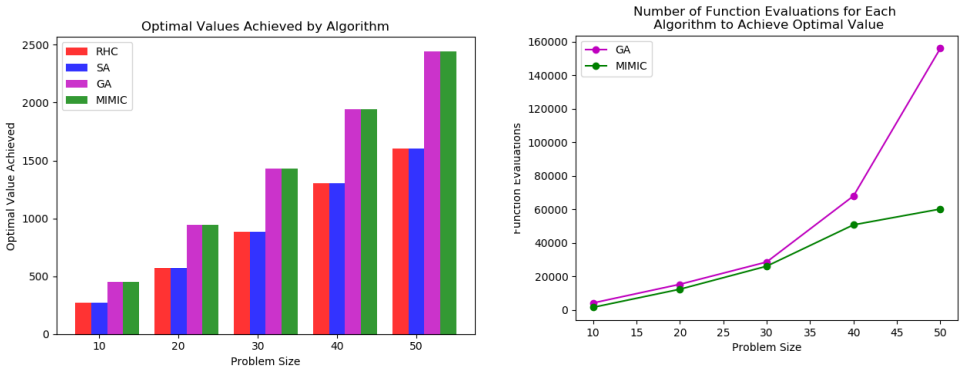
A graph is K-Colorable if it is possible to assign one of k colors to each of the nodes of the graph such that no adjacent nodes have the same color. (Isbell, 1996). A

good evaluation function for this problem would then be a function that scores an instance of a graph g higher than g' if g contains more nodes whose adjacent nodes are different colors.

$$h(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

$$f(g) = \sum_{i=0}^{n} \sum_{j=0}^{v_i} h(color(g_i), color(g_{ij}))$$

Where n is the number of nodes in a graph and v_i is the number of adjacent nodes for the i^{th} node. g_i is the i^{th} node in graph g and g_{ij} is the j^{th} adjacency node of node i in graph g. The *color* function returns the color of a specific node. Varying the algorithm hyper-parameters had little significance, so the figure below plots one optimal value per algorithm per problem size.



(a) Optimal values achieved by each algorithm on the max k-coloring problem

(b) Genetic and MIMIC function evaluation plot to distinguish the performance of each algorithm

Figure 3: *Subfigure (a) shows the comparison of optimal values achieved by different algorithms on the max k-coloring problem. Subfigure (b) shows the comparison of function evaluations between GA and MIMIC.*

Mimic and GA both work well on the max k coloring problem as they are both able to consistently find a solution to each problem instance given the problem size. However, MIMIC clearly outperforms GA in terms of the number of function evaluations needed to achieve the solution to the problem. GA shows an almost exponential growth in the number of functional evaluation given the size of the k coloring problem, while MIMIC only shows a linear increase in the number of function evaluations needed to find a solution for the problem. MIMIC and GA perform better than SA and RHC on the max k coloring problem because both algorithms encode some kind of history into future populations. This concept is important in the k coloring problem because there are cases where the whole problem can usually be solved by solving smaller instances of the problem and then combining each smaller instance into the larger problem. Algorithms that maintain some form of history do a better job at first solving smaller instances of the problem and fitting those solutions into the entire problems.

2 Optimization in Neural Networks

The goal of the previous optimization problems was to find a solution $x \in \mathbb{Z}^n$, such that $\forall x' \in \mathbb{Z}^n, f(x) > f(x')$, where f was defined to be an evaluation function

($f : \mathbb{Z}^n -> \mathbb{R}$) for the specified problem. These types of problems are referred to as discrete optimization problems because the components of the solution space are restricted to be integer values. Optimization problems can be transformed into continuous problems by changing the definition of the evaluation function f to be $f : \mathbb{R}^n \rightarrow R$. This inherently changes the set of all possible solutions to be $\mathbb{R}^n$.

One example of an optimization problem involves finding the optimal weights $w \in R^n$ (one hidden layer for simplicity) of a neural network, such that w minimizes the error in the model. Typically, neural networks optimize w by using back-propagation with stochastic gradient descent (SGD) to iteratively decrease the loss of a model. In addition to SGD, the algorithms that were analyzed previously (RHC, simulated annealing, and genetic algorithms) can be used to optimize w. This sections compares SGD to randomized optimization algorithms by training several neural networks with different convergence algorithms on a financial dataset. The goal of the network is to classify whether a potential loan will be good or bad. The following log loss cost function was used to calculate the error in the model:

$$C(x) = -1/n \sum_{i=0}^{n} y \log(p_i) + (1 - y) \log(1 - p_i)$$

The evaluation function was then defined to be:

$$f(x) = -1 * C(x)$$

The loss is multiplied by negative one in the evaluation function because the goal of each optimization algorithm is to maximize the value produced by the evaluation function. This is done because maximizing the negative log loss is similar to minimizing the log loss, of which the latter is done in a standard back-propagation algorithm.

2.1 Randomized Hill Climbing

Figure 4 shows the optimal values found by the randomized hill climbing algorithm vs the number of iterations the algorithm ran for. The algorithm was run using five random restarts for a maximum of 5000 iterations per run. Each run was also terminated if the algorithm failed to find a random neighbor with a higher optimal value for fifty attempts in a row. The figure clearly shows a steady increase in the optimal values per run, but it can also be seen that as the optimal value increases, the number of failed attempts to find a neighbor with a higher optimal value increases. So, the algorithm could possibly improve on the best solution found by increasing the number of maximum attempts before failing and the number of maximum iterations per run. However, the optimal value increase may only be slight because the number of failed attempts increases as the optimal value increases. This in turn would cause the algorithm to run longer and so the tradeoff might not be worth increasing algorithm's hyperparameters.

2.2 Simulated Annealing

Figure 5 shows the optimal values found by the simulated annealing algorithm vs the number of iterations the algorithm ran for. The plot is actually quite similar to a single RHC plot in terms of how fast the optimal value increases given the number of iterations the algorithm runs for. The RHC and SA algorithms were run with the same number of hyperparameters in terms of maximum iterations and maximum attempts, so that explains the similar growth rates of the two algorithms. However,

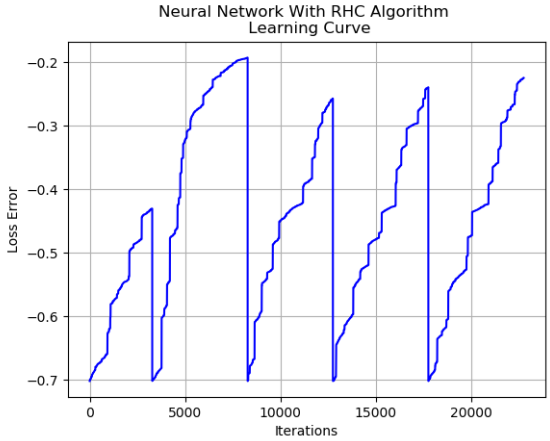


Figure 4: *Optimal value achieved as a function of iterations for the RHC algorithm.*

the SA algorithm used additional hyperparameters that controlled the probability in which the algorithm changed the neural network weight vector, so that the weight vector actually produced smaller optimal value. This seems counterintuitive to the goal of the search algorithm, but it actually prevents the weights w from getting stuck at local maximum values. The temperature and tempature decay rate are the hyperparameters that control the probability in which the algorithm selects values of w that result in a lower optimal value. A high temperature increases the probability that the algorithm chooses a less optimal w, and the decay rate reduces the temperature each iteration, so that the probablility of choosing a less optimal w grows smaller the longer the algorithm runs for. This explains why the plot in figure 6 grows quickly at first, and then slows down as the number of iterations increase. Another similarity between SA and RHC is in the way the algorithms could be improved. The algorithms could improve by setting higher hyperparameter values for the maximum number of iterations and maximum number of attempts that control when the algorithm terminates. However, doing so will result in longer run times and possibly only a slight increase in the optimal value achieved.

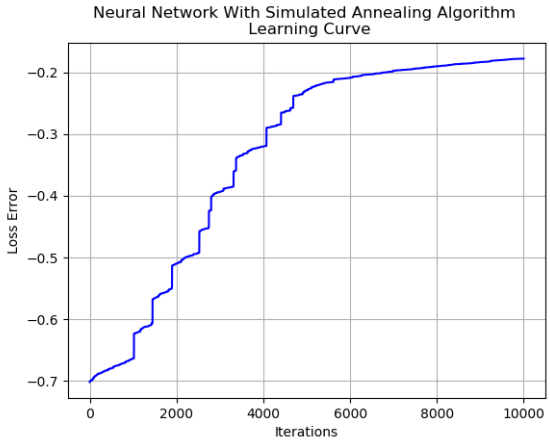


Figure 5: *Optimal value achieved as a function of iterations for the SA algorithm.*

2.3 Genetic Algorithm

Figure 7 shows the optimal values found by the genetic algorithm vs the number of iterations the algorithm ran for. The genetic algorithm performed the poorest in runtime and in finding an optimal weight vector w for the neural network classifier

in comparison to the other algorithms analyzed in this paper. Each iteration took significantly longer than the other algorithms because the genetic algorithm performs the "natural selection" step each iteration. The natural selection step will take longer to run for higher population sizes. The population size was set to 250 for this problem and the mutation rate was set to 0.99. The algorithm performed worse if the population or the mutation rate was set to lower values. The algorithm performed slightly better for higher population sizes, but at the cost of much longer run times. The population size of 250 was chosen because it allowed the algorithm to complete in a reasonable amount of time and find a value for w that is very close to the w that would have been found with a larger population size. One nice property of genetic algorithms is that they can be made more effiecient by parrallizing the algorithm. Therefore, if one has access to a machine with multiple processors, the population could be increased without the cost of longer runtimes.

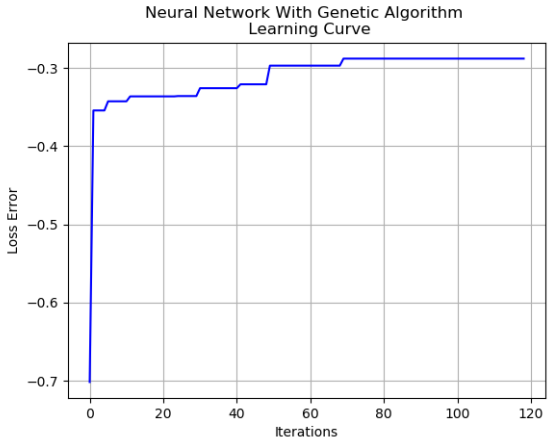


Figure 6: *Optimal value achieved as a function of iterations for the GA algorithm.*

2.4 Stochastic Gradient Descent

Figure 7 shows the optimal values found by the SGD algorithm vs the number of iterations the algorithm ran for. The optimal values increase faster and more smoothly in comparison to the previous algorithms. This is because stochastic gradient descent uses back-propagation to update the neural network weights each iteration. During each iteration, the algorithm computes an error gradient vector that defines how each component of w is updated. Therefore, each component of w moves closer to the global optimum weight vector each iteration. This is in contrast to the other algorithms that move each component of w in a random direction each iteration. This explains why stochastic gradient descent is more efficient than the other optimization algorithms.

Table 2 compares the algorithms over several performance metrics and SGD shows better results for each category. Simulated annealing shows to be the best randomized optimization algorithm as it obtained the next best scores in all categories. The genetic algorithm required significantly less iterations to obtain an evaluation score higher than -0.3 compared to SA and RHC, but the algorithm was not able to improve past and evaluation score of -0.28. RHC compared very closely to SA, but restarting the algorithm over separate runs caused RHC to run much longer than SA. The randomized optimization algorithms shown in this section provide insight into different ways in finding optimal weights for a neural network.

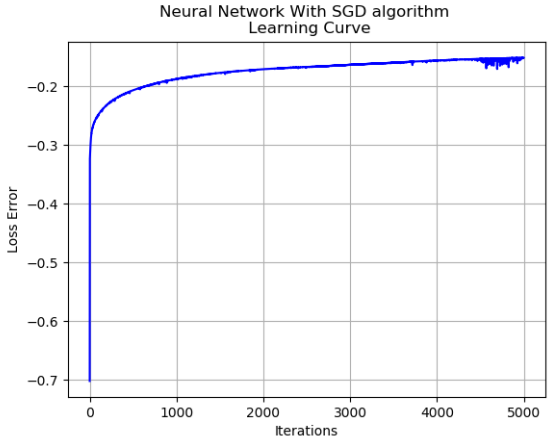


Figure 7: *Optimal value achieved as a function of iterations for the SGD algorithm.*

Algorithm	Training Time	Training Loss	Test Loss	Training Accuracy	Test Accuracy
RHC	713	0.193	0.219	0.933	0.925
SA	601	0.178	0.209	0.937	0.932
GA	156	0.322	0.347	0.896	0.885
SGD	289	0.151	0.161	0.952	0.951

Table 3: General metrics regarding each optimization algorithm that was used to train a neural network classifier to predict good or bad loans

3 Conclussion

The goal of this paper was to show how different randomization algorithms compare against one another when they are used to solve different types of problems. The experiments conducted in this paper would not have been possible if it weren't for the open source projects ABIGAIL and mlrose. ABIGAIL was used to analyze the three discrete problems from the first section and mlrose was used to analyze the different neural networks. Thanks to everyone involved in those projects.

References

[1] Hayes, G. (2019). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python. https://github.com/gkhayes/mlrose. 8/10/2019.

[2] Kolhe, Pushkar (2018). ABIGAIL. https://github.com/pushkar/ABAGAIL. 8/10/2019

[3] Isbell, Charles. (1996). Randomized Local Search as Successive Estimation of Probability Densities a. https://www.cc.gatech.edu/ isbell/tutorials/mimic-tutorial.pdf. 8/10/2019.