Comparison of Four Randomized Optimization Algorithms

Brian Ambielli
Graduate Student, Georgia Tech
https://github.gatech.edu/bambielli3/ML-2

ABSTRACT

This paper explores the performance of 4 different randomized optimization (RO) algorithms – Random Hill Climbing, Simulated Annealing, Genetic Algorithms, and MIMIC – in the context of 4 different optimization problems – finding optimal weights for a neural network, \`countones\`, \`fourpeaks\`, and \`knapsack\`. These problems were selected to highlight the strengths and weaknesses of each of the randomized optimization algorithms explored in this paper. The data collected also showed that backpropagation is a better method for optimizing the weights of a neural network than RO algorithms are, both in the accuracy of the trained network and the necessary training time.

KEYWORDS

Learning and Training;

INTRODUCTION

This paper compares 4 randomized optimization (RO) algorithms in the context of different problem scenarios. Each scenario was designed to outline the strengths and weaknesses of each RO algorithm. The 4 random optimization algorithms that will be explored by this paper are random hill climbing (RHC), simulated annealing (SA), genetic algorithms (GA), and MIMIC. Each RO algorithm has distinct strengths and weaknesses, which are highlighted by the problem scenarios chosen.

METHODS

The first 3 problem scenarios were designed to highlight the strengths of SA, GA and MIMIC. SA excels when the fitness function \`f(x)\` is computationally inexpensive. SA also has a higher chance of finding the global maximum when there is a \`large basin of attraction\` that will inevitably lead to the maximum. These properties of SA will be highlighted by the \`countones\` problem, which was designed to have a single maximum with a small but increasing slope towards that solution. The goal of \`countones\` is to maximize an evaluation function which penalizes the presence of 0's in a bit string. At each step an f(x) evaluation indicates how many bits are set to 1. The goal is to maximize the value, which should lead the algorithms towards the global optima where the bit string contains all 1's. This problem has only one global optimum, and has a large basin of attraction towards that optimum, which should cause SA to excel. To collect more data about RO algorithm performance in this context, I will vary the size of the bit string \`N\` over the range 40 – 200 bits.

The second problem I will explore in this assignment will be \`fourpeaks \`, which is designed to show off the strengths of GA. The problem itself was created by Shumeet Baluja and Rich Caruana, and was designed to specifically highlight the strengths of GA outside of a traditional 'biological' context[2]. GA excels when f(x) is quick to compute, but also when it can exploit some level of "locality" within the problem structure. There are two optimal solutions to the fourpeaks problem, one when the bit-string being explored starts with only 1's and ends with the rest 0's, and the other optima is the inverse. There are also two local optima that are lower in value (when the bit string is either all 1's or all 0's) which were designed to trap simpler hill climbing approaches like SA with their large basins of attraction. The fitness function f(x) is parameterized by a value T, which stretches or shrinks the basins of attraction on the global optima. I chose a value of \`T=N/5\` where N is the size of the bit string being explored. I also chose N values that range from 20 to 100. This combination of N and T will result in large basins of attraction for the two suboptimal peaks, and relatively small basins for the global optima, which should trap SA / RHC. Finally, the structure of the fourpeaks problem lends itself to a single point of crossover in the crossover phase of GA, since the optimal solution has a sharp transition from 1 to 0. Because of this background knowledge, I used a \`SingleCrossoverStrategy\` in my GA implementation. The small basins for the optimal solutions, and the resiliency of genetic algorithims against getting stuck in local optima as much as hill climbing, makes four peaks a good candidate to highlight the strengths of GA when compared to SA and MIMIC.

I chose the `knapsack` problem to highlight the strengths of MIMIC. `knapsack` is a notorious NP-Hard problem in computer science that has to tractable solution for finding the global optimum. The problem presents the solver with a set of items of varying weights and values, and a maximum allowed weight for the "knapsack". The solver must then produce an optimal selection of the presented items, such that the combined weights are below the maximum allowed weight and the combined value of the chosen items is maximal. Instead of randomly bouncing around between selections of items (which is the approach of RHC, SA, and GA) it would be good for the solver to "learn" something about the underlying distribution of possible solutions after each round to minimize the number of f(x) evaluations in future iterations and prevent re-exploring known bad solutions. This is exactly the approach of MIMIC, so I'm hoping that the knapsack problem will show off its relative strength in this regard in comparison to SA and GA. To see how the RO algorithms perform in the context of this problem, I will be varying the number of items to choose from in the knapsack `N` from 40 – 200.

For each of `countones`, `fourpeaks`, and `knapsack`, I collected data about the number of f(x) function evaluations required for each algorithm to complete, the time it took to train each algorithm, and the value of the optimal solution it came up with. Since each of these 3 problem scenarios involved maximizing the fitness function, a higher optimal solution is always better. Each of the algorithms (RHC, SA, GA, and MIMIC) were plotted against each other on each of these different measurement scales. The Y axis in the plots represents one of the metrics I captured about algorithm performance, and the X scale represents the size of the problem space, which varied between runs. The number of function calls was collected by adding a counter to each of the RO class files, which was incremented every time the algorithm performed an `eval.value()` calculation. To get a true number of function calls necessary to return an answer, it was important to also count function calls that are necessary to return the optimal value after training is complete (i.e. in mimic.get_optimal()).

For the final scenario, I compared how RHC, SA and GA perform in the context of finding the optimal weights of a neural network to model the `Minimal_Derived` dataset that I used from assignment 1. The neural network I trained in Assignment 1 used backpropagation + gradient descent to fit weights to the neural network instead of an RO method. My expectation is that choosing to use an RO algorithm instead of backpropagation will result in a neural network that is less accurate, as these algorithms have the tendency to get stuck in local optima and are randomly exploring the solution space instead of getting feedback propagated back to them at each iteration.

The implementations for each of the four scenarios analyzed in this assignment were borrowed from the ABAGAIL[1] library by Pushkar Kolhe. ABAGAIL is a java library with jython adapters, that contains implementations for the algorithms and the problem contexts I will be exploring in this paper. I adopted scripts from ABAGAIL for my own experiments, you can find them in the repository associated with this paper (see ./jython/).

To see the results, clone the repository and follow the steps outlined in `README.md`. This will collect data for each experiment, and generate the plots discussed below, in the `./data` folder.

RESULTS

Find data + discussion of results for the 4 experiments below:

countones

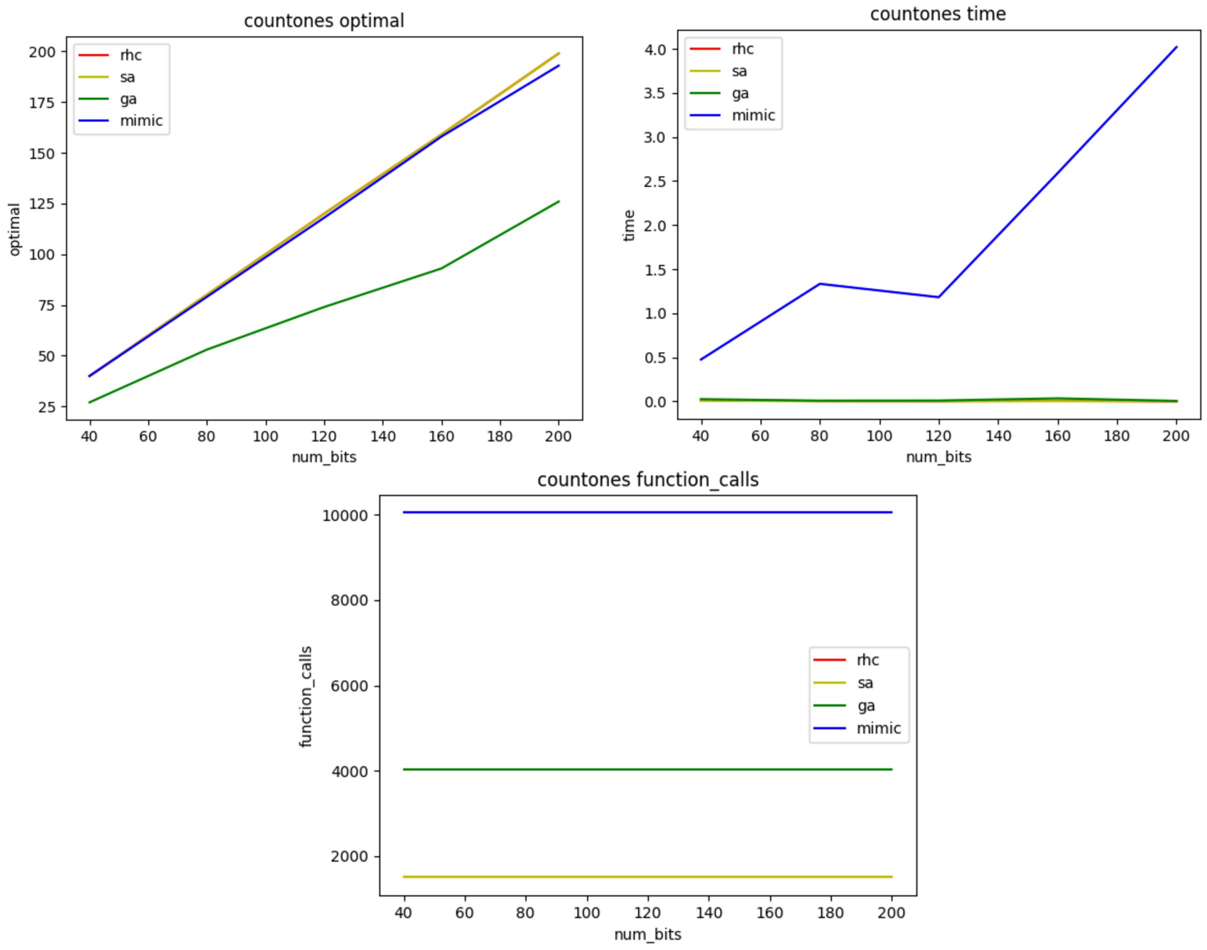

Figure 1: countones experiment data (note: time is in seconds, and is reflective of training time)

Looking at the "countones optimal" graph alone, it is clear that SA / RHC are better than GA at finding the optimal value, but it isn't entirely clear that SA / RHC are that much better than MIMIC. It seems though that as the number of bits in the problem space gets larger that the divergence between MIMIC and SA would be greater, so if I were to rerun the experiment perhaps I would try to increase N to some factors larger each time to exaggerate this difference. The real benefits of SA / RHC over MIMIC are apparent in figures "countones time" and "countones function_calls". Where MIMIC needed 10k f(x) function calls to achieve the accuracy that it did, SA / RHC only needed 1500. This was due to the simplicity of the problem space: there really wasn't anything useful or necessary to "learn" about the underlying distribution in the problem space, so the additional work that MIMIC does to reduce the H space in future iterations was largely unnecessary. It is also clear from the "countones time" graph that SA / RHC were also superior in the amount of training time necessary: mimic took 4.5 seconds on the N=200 instance, where SA took .002 seconds. That's over a factor of 3 more time necessary to finish training. When considering the balance of `exploration` vs `exploitation` to find global optima in a distribution, countones was clearly designed to be a pure `exploitation` scenario (one single global optima with a basin of attraction that spans the entire problem space). SA / RHC (especially RHC) are simple algorithms that are most successful when `exploitation` of neighbor knowledge in a problem space leads to the optimal solution, so it makes sense that they would excel in the context of countones. If only all problems could be this well behaved, we could just use SA / RHC to solve all our problems!

fourpeaks

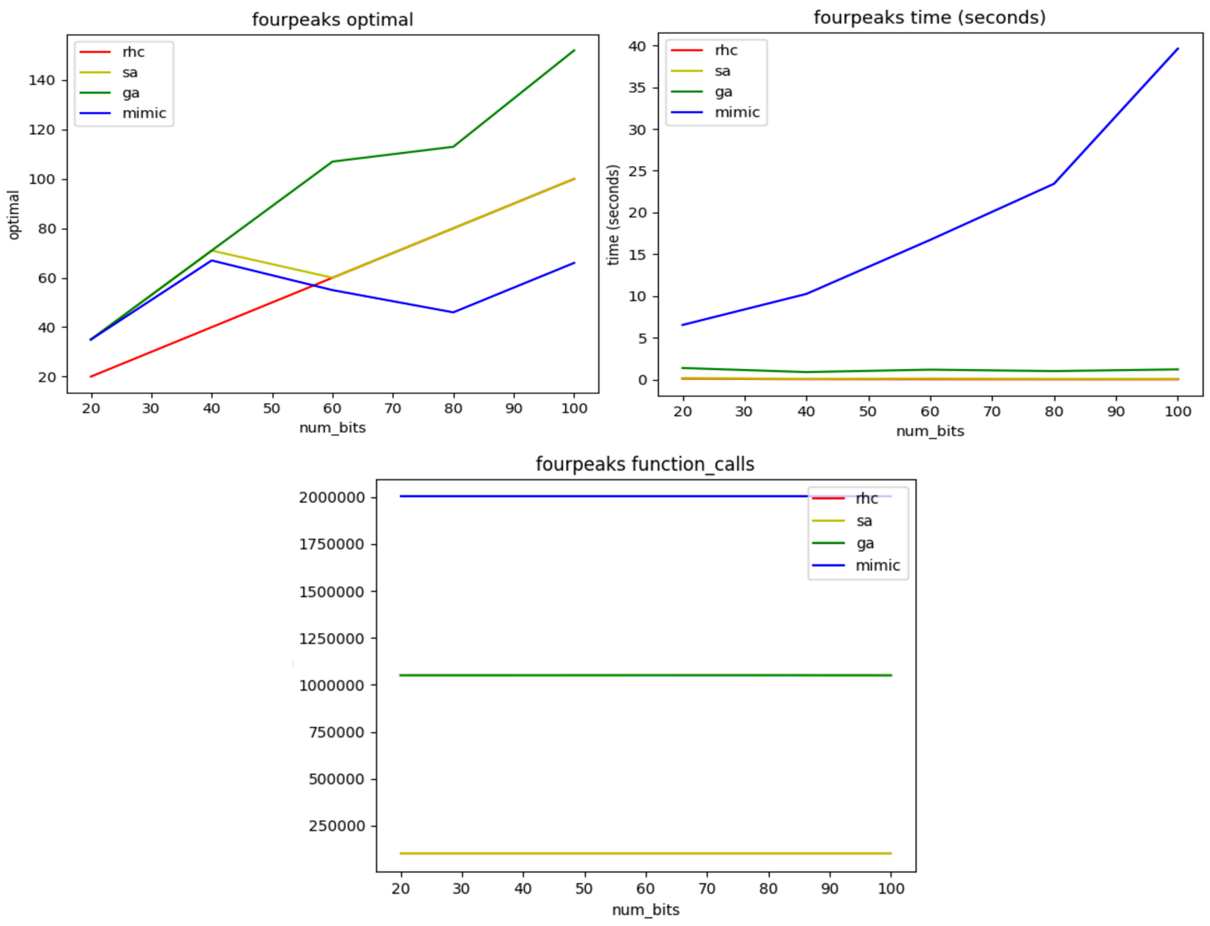

Figure 2: fourpeaks experiment data (note: time is reflective of training time)

The fourpeaks problem space was designed to show off the strenghts of GA, and it is clear from looking at "fourpeaks optimal" graph that GA wins out over the other algorithms in terms of producing the most optimal solution for fourpeaks. As a reminder, the f(x) fitness function for fourpeaks penalizes solutions that do not have a sequence of 1s directly followed by a sequence of 0s, both of which have a length greater than the bound of T set by the problem designer. In my case, T was always N/5, so in the case of a 5 bit sequence, the optimal solution would be `11100` or `11000`. These global optima act as "peaks" with small basins of attraction, making them difficult to encounter as SA / RHC. When N is relatively small (like in the case where N=20 in the experiment I ran), it is still reasonably likely that over the course of 100k iterations (which was the numbr chosen for SA / RHC for this experiment) SA would stumble into the basin of attraction for these small peaks and find the optimal value. As N became large, though, the likelihood of this occurring decreased substantially, and it is apparent from the `forupeaks optimal` graph that SA / RHC were just able to find the local optima "traps" as they were only able to achieve scores proportional to the size of N (which is the highest score you can achieve from the local optima). GA is able to overcome this deficiency. The high degree of locality in the optimal solution (sequence of 1s followed by sequence of 0s) mirrors a single point crossover in GA. With large enough populations, it is likely that GA will "breed" a solution that reaches one of the optimal peaks with this crossover strategy. A multi-point crossover strategy might not have been so successful, so choosing single point crossover for GA is an instance of tuning the algorithm's parameters with prior knowledge about the problem space.

GA ran an entire factor higher of f(x) evaluations when compared to SA / RHC: 1 million vs 100k. The time it took to complete the GA evaluations also took one factor more time than SA / RHC: 1 second vs miliseconds. MIMIC, while trying to learn something about the structure of the underlying distribution, began to exhibit exponential-like time complexity as N increased. GA did not exhibit this: while it took longer to run GA than it took to run SA / RHC, it still appears to converge in a constant amount of time. This is good news for the scalability of GA in this problem space. Since finding the optimal solution is always our goal, GA is the clear winner based on the data collected for fourpeaks, since it returned the best solution in a constant amount of time as N increased.

knapsack

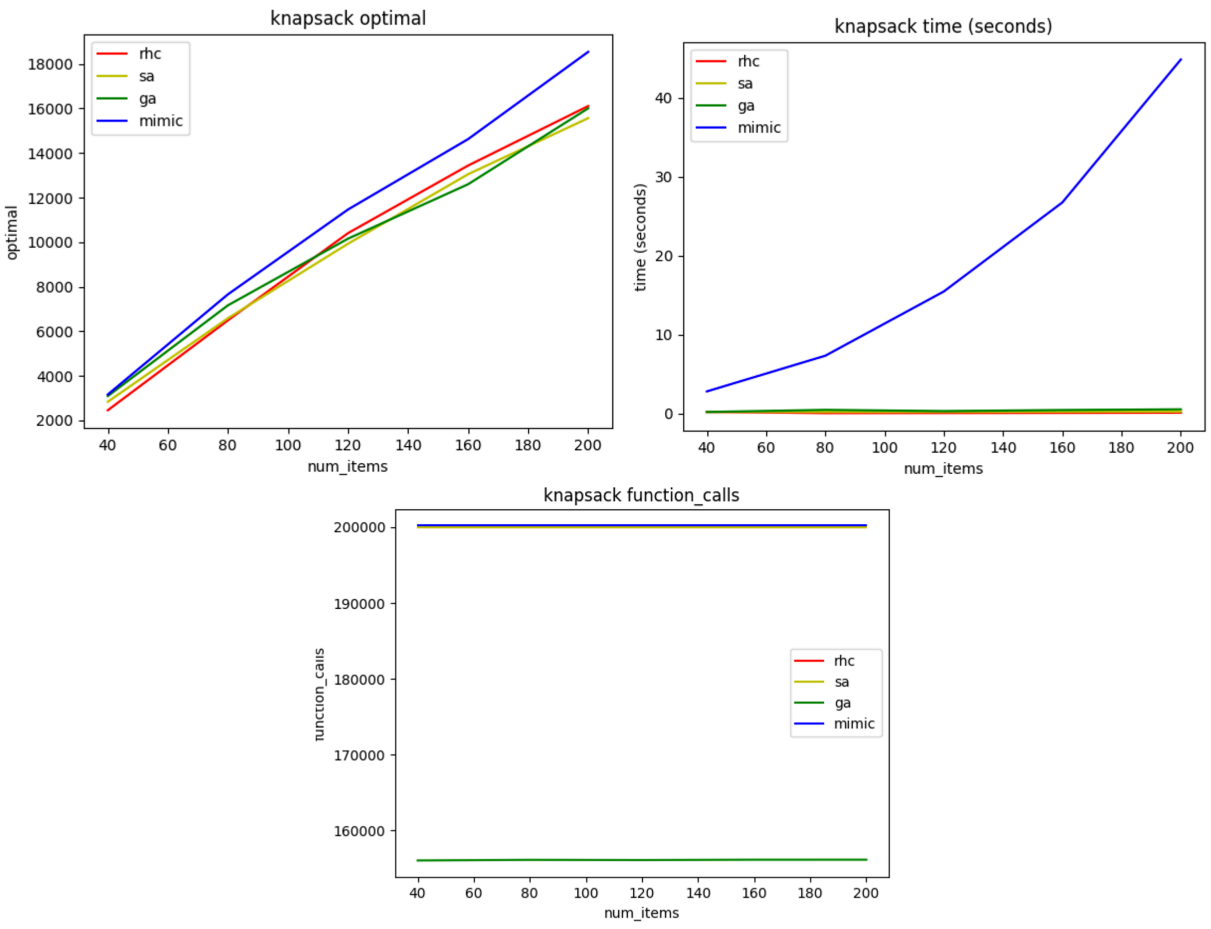

Figure 3: knapsack experiment data (note: time is reflective of training time)

The knapsack problem is NP-Hard and finding an optimal result has no known tractable solution: applying randomized optimization methods is the best we can do. Of the RO algorithms explored in this context, MIMIC was consistently able to return the most optimal solution, but at the sacrifice of a longer amount of training time. MIMIC attempts to learn something with each iteration about the underlying distribution that it is exploring. As N increases and the distribution of item subsets becomes larger, it makes sense that MIMIC would take longer to return its optimal value. MIMIC even looks like it might be running in exponential-like time in relation to N, which is important to note if you were trying to solve knapsack with time constraints.

As MIMIC learns about the underlying distribution, it eliminates sub-optimal areas of the solution space. This allows mimic to focus on exploring higher potential areas in subsequent iterations. In other words, the knowledge MIMIC learns about the solution space allows it to return strictly better solutions with each iteration, until it eventually converges on a global optimum or runs out of iteration time and returns the best it has found so far. Since the solution space for knapsack is so large, with many small basins of attraction for local optima throughout, SA / RHC / GA have a higher likelihood of getting trapped in local optima or just never exploring the area of the solution space that has the true global optimum. There is no underlying "locality" in the solutions of this problem space that GA can exploit either, particularly with single point crossover. MIMIC is superior when working in solution spaces where learning about the underlying distribution is beneficial. In the case of knapsack MIMIC is able to hone in on "high potential" areas of its large solution space, preventing future iterations of the algorithm from exploring low potential areas again as the distribution is refined with each iteration.

Increasing iterations of SA / RHC wouldn't necessarily guarantee reaching the global maximum, since random restarts could continually fall in basins of attraction to local optima. When the solution space is large (like in the case of knapsack) a solver like MIMIC that actually reduces the solution space to explore with each iteration is preferable, when the goal is to find the closest thing to the global optimum, not just any local optima.

NEURAL NETWORKS

Find results from running the Neural Network experiments below. `Minimal-Derived` data set can be found in the folder `./data`. Both runs involved setting the input nodes to 7644 (to match the number of attributes in the one-hot-encoded minimal-derived data set) and a single output node (since a single classification value was desired). The hidden layer nodes were varied between the first and second runs, from 1 node, to 50 nodes. Two layers of 50 nodes seemed to provide the best results in the backpropagation experiment from assignment 1, but it does not appear that ABAGAIL supports more than 1 hidden layer at a time. I tried running the experiment with 100 hidden nodes, but ran in to memory constraints on my machine with the GA solver.

Algorithm	Number of Hidden nodes	Correctly Classified	Incorrectly Classified	Training Time (s)	Testing Time (s)	Percent Correctly Classified
RHC	1	1607	346	17.683	1.139	82.284
	50	1632	321	280.816	13.255	83.564
SA	1	1297	656	20.254	0.993	66.411
	50	321	1632	291.788	14.357	16.436
GA	1	1632	321	915.752	1.055	83.564
	50	1633	320	52035.481	15.321	83.615

Figure 4: Data collected from Neural Network Experiment

Note that the classes in my dataset are heavily skewed towards the "false" case. In fact, there are exactly 1607 false classes in my data set. Therefore even though it appeared that RHC and GA trained a reasonable network, with an accuracy of around 83%, this result is actually not much better than a naïve classifier that just predicts that everything is "false". In assignment 1, when a neural network was trained on this data set using backpropagation with gradient descent, the resulting classifier achieved 96% accuracy on the test set.

I wouldn't expect these RO techniques to perform better than backpropagation when training a neural network. Backpropagation feeds information back to the network on each iteration, allowing it to gradually tune its weights to continuously improve its classification accuracy based on current performance. The same feedback mechanism does not exist with a randomized optimization approach.

Neural networks also tend to fall in to local optima of weight assignments during training. This is why initializing a network's weights to small random values is important, to try and force future training iterations to explore other portions of the solution space. It was clear that the weights for my network were falling in to a local optimum during SA / RHC training in these experiments, since the error value returned after each iteration was *identical*. GA, by design, will not get stuck in these local optima, but since there is no structure or locality to the optimal solution for the algorithm to exploit, so it also will not perform better than backpropagation with gradient descent.

Training time was also poor for RO algorithms in comparison to the NN trained with backpropagation from assignment 1. The most accurate classifier I was able to train in assignment 1 converged after 42 seconds (adam + relu + (33,33,33) hidden layers) while a single node hidden layer for RHC was able to converge after 17 seconds but did not do any better than a naïve classifier. GA took 14 hours to complete training. Part of this training time discrepancy could be due to the implementation of NN in ABAGAIL vs sklearn, so it isn't necessarily that strong of a comparison to the networks trained in assignment 1.

Based on the data collected from these experiments, my initial intuition seemed to be correct: none of the RO methods used to train neural networks would consistently be able to achieve better results than backpropagation with gradient descent.

CONCLUSION

The experiments above were designed to highlight the strengths and weaknesses of SA / RHC, GA, and MIMIC. The results from these experiments made it apparent that knowledge of the problem space is critical when choosing which RO algorithm to use to find an optimal solution. If there is no "structure" to the underlying distribution of the solution set, you should choose SA / RHC since a random walk is about as good as you will be able to get. If there is some structure to your solution space (e.g. many local optima with small basins of attraction that lead to global optima) then choosing something like MIMIC should be better so you don't waste time continually exploring bad areas of the problem space during future iterations. If there is some degree of locality between attributes in your solution space, the evolution-like crossover of genetic algorithms might get you to the optimal solution. Finally, the Neural Network experiment cemented the notion that the feedback mechanism of backpropagation with gradient descent is the best way to train the weights for a neural network, and that RO algorithms don't have a chance of competing with this gold standard.

CITATIONS

1. https://github.com/pushkar/ABAGAIL
2. Baluja and Caruana 1995
   https://pdfs.semanticscholar.org/cd4f/e89d8dd6060e2957041f90fc699a30058d01.pdf