Randomized Optimization (March 2017)

William Z. Ma, *Student, Georgia Institute of Technology*

Abstract – This document gives detailed analysis of four randomized optimization algorithms: randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC. The analysis will be performed in two parts. First, a familiar scenario: We will take the neural network that we optimized in our analysis of supervised learning algorithms and attempt to replace backpropagation with one such randomized optimization algorithm. Second, we will present three optimization problem domains, and analyze the use of the four algorithms on each domain, highlighting any potential advantages and disadvantages of each.

1 Randomized Optimization and Neural Network Weights

1.1 Introduction and Recap

In our last paper, we analyzed the performance of various supervised learning algorithms and tuned their parameters for a dataset. Specifically, we used the results of Free Code Camp's 2016 New Coder Survey, a dataset containing a wide array of information about prospective coders, including such information as age and socioeconomic background. The question we then asked was whether a particular gender of prospective coder was predominantly of a certain age group or ethnicity, and thus we attempted to predict a prospective coder's gender based off of their age, income, whether they were an ethnic minority, and whether they were already a software developer.

We then analyzed the effectiveness of various supervised learning algorithms such as k-NN, and neural networks, in particular multi-layer perceptrons. For neural networks in particular, we identified the optimal values of certain hyperparameters, which we will use again in this paper. The relevant hyperparameters for the neural network are the following:

1. Maximum number of nodes in a hidden layer: **18**
2. Maximum number of hidden layers: **14**

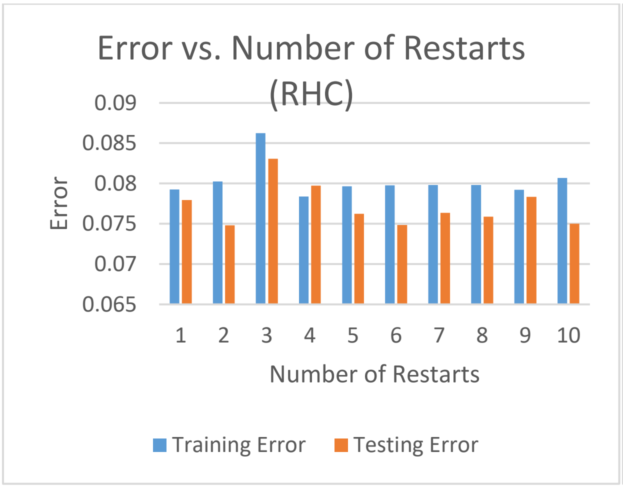
As it happens, such neural networks remain the topic of interest, as there are a variety of ways for such neural networks to update their weights, one of which is backpropagation. However, we can in fact replace this backpropagation with a randomized optimization algorithm. We will use the optimal hyperparameter values we found for our neural network, so that we may thereby directly compare the performance of our various randomized optimization algorithms. In this manner, we will have not any contravening variables for our analysis. Before we begin our analysis, allow us to note the following invariants about our analyses:

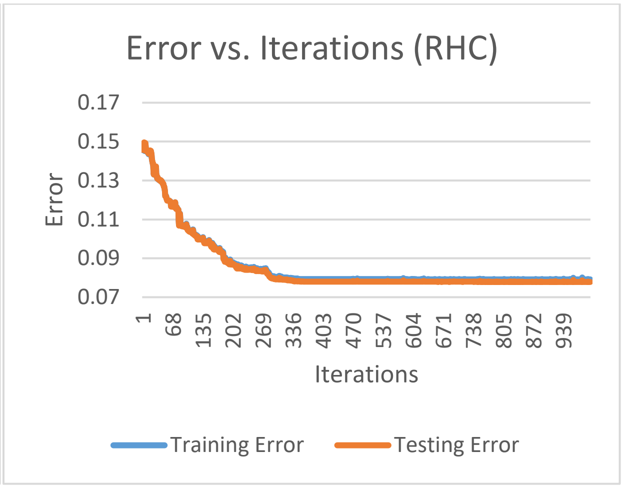
- Error always refers to the mean squared error (MSE)
- All algorithm implementations are those that are included in the ABAGAIL package
- All the data was pre-normalized to a Gaussian distribution using the Pandas Python package
- There is no cross-validation data available; although excellent for analyzing overfitting and generalizing algorithms to data we have yet to see, ABAGAIL simply doesn't have the functionality available

1.2 Randomized Hill Climbing (RHC)

Randomized hill climbing is the simplest randomized optimization algorithm, consisting of starting at a random point, looking at the fitness of the neighboring points, going to the point with the best fitness, looking at the fitness of the neighbors of that point, and proceeding to the neighbor of that point with the best fitness. The algorithm then proceeds in this manner for each iteration, until it finds a point where no such neighbors give better fitness. It is at this point that we have either reached the goal of the function, either the highest fitness point or the lowest cost point, or we have become stuck at a local optimum.

There is only one potential hyperparameter for randomized hill climbing, and that is the number of restarts that the algorithm undergoes, trying to pick the best results between the various times the algorithm is run, in case a specific run gets stuck at a local optimum:





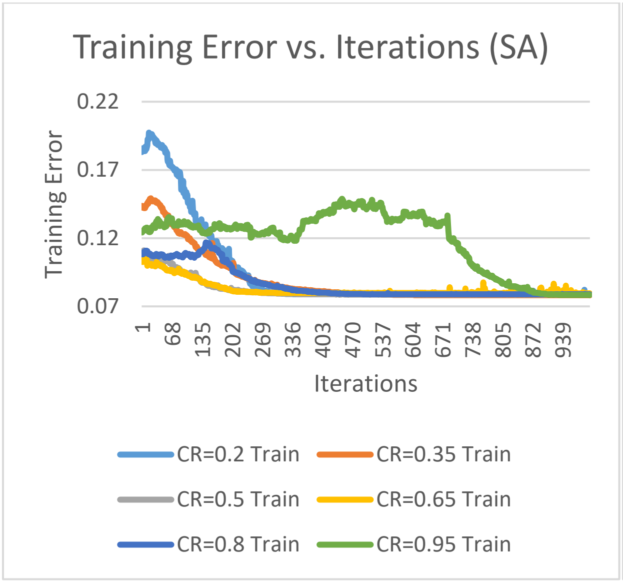
We can see pretty clearly above that for a normal run, the error converges after about 300 or so iterations, to a final error of 0.079. While perhaps unimpressive by number of iterations, remember that RHC is by far the fastest algorithm used in this paper; therefore despite taking perhaps even 5x more iterations than some other algorithms, it will almost certainly still finish first, if taken in a race. However, we are also aware that a randomized hill climbing algorithm may be prone to getting stuck in local optima. For this reason, we attempt to determine how many restarts it might take to mitigate this. We can see that we in almost all cases will need only 1 restart in order to get the best performance, on the basis that we will almost always converge to a global minimum.

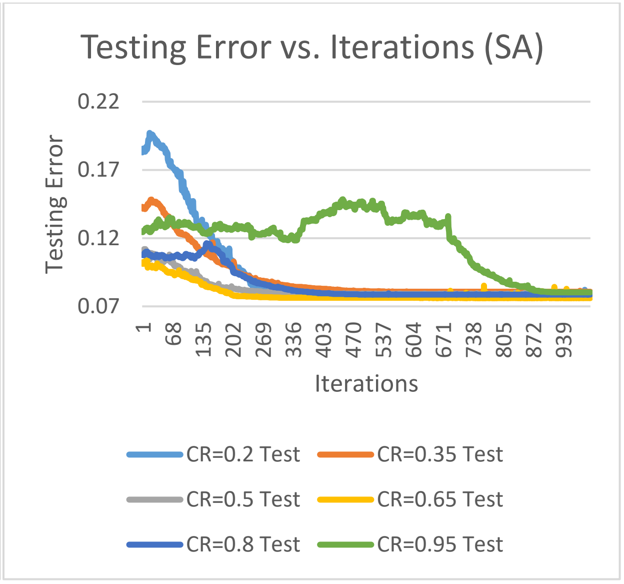
1.3 Simulated Annealing (SA)

Simulated annealing is a randomized optimization algorithm inspired by the concept of annealing in metallurgy, whereby a metal is heated up and cooled down in a controlled manner to allow the molecules in the metal to align, increasing the overall strength and quality of the metal. The algorithm takes this general idea and extends it to data, using several concepts from probability theory and statistics.

The general algorithm begins at a random point. It then begins its first iteration by sampling a random point from elsewhere in the data. The distance between these two points, as would typically be defined by some distance or neighbor function, is instead based off of a probability distribution, such as the Boltzmann distribution, often used to model energy distributions of large systems of particles. This distance is then considered towards some goal, such as increasing our fitness function or lowering a cost function. The algorithm then decides whether or not to go to this point, depending on whether it moves us toward the goal. However, since data distributions are often filled with local optima that are some distance away from the goal, the algorithm may decide to move to a point farther from the goal in order to avoid becoming stuck in one of these local optima, conferring a substantial advantage to the algorithm over randomized hill climbing, which is prone to getting stuck in such regions; at least in theory.

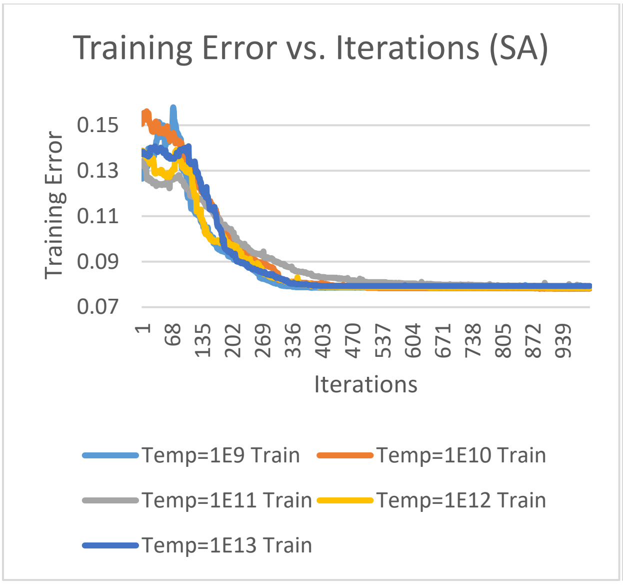
True to the theme of heating up metal and cooling it in a controlled manner, there are two hyperparameters of concern when it comes to simulated annealing: cooling rate (CR), and starting temperature (Temp). We begin with an analysis of cooling rate on the algorithm. Cooling rate is a constant, anywhere from 0 to 1, defining how quickly the temperature is decreasing with each iteration (lower values cool faster). This decrease in temperature corresponds to a decrease in how much the algorithm can jump between points; that is to say, the lower the temperature at any iteration, the more the algorithm is inclined to move towards points that move us closer toward the goal, increasing the chances of getting stuck in a local optima. The general idea of the algorithm then, is that we sample many random points in the beginning, in an attempt to determine a best optimum to move towards, slowly accelerating towards this optimum, and localizing the space of available points as the algorithm continues. But what value is optimal for cooling rate? We picked 6 different values and tried them on our dataset:

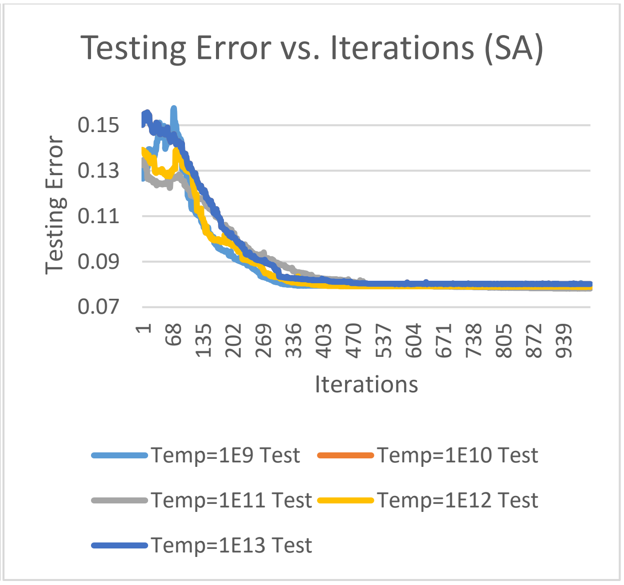




The above represents, for each value for our cooling rate, the average error progression over 10 trials of 1000 iterations each. One thing we notice immediately is that no matter which value for the CR we chose, that all of them eventually converged to the same optimal error. However, this does not mean that any value for the CR is admissible. For practical purposes, we'd like a value that reaches the optimum in the fewest number of iterations, thereby saving us time. What else? We'd also like a value that consistently performs better than any other value, if one such value exists. As it happens, it seems that a value of CR=0.65 is able to match both these criteria. It not only reaches the optimum that the others do, it reaches that optimum much more quickly than the others; as well as having consistently lower error than all other values. For this reason, we pick 0.65 as the optimal value for CR.

We also mentioned another hyperparameter, namely the starting temperature; the starting temperature's effect is, perhaps intuitively, only important for the first few iterations, its effect becoming substantially less pronounced as more iterations occur; a result of the cooling rate. So how important do we need our starting iterations to be? We picked 5 starting temperatures, two orders of magnitude below and above the default for the algorithm, and tested them on our dataset, again, averaged over 10 trials of 1000 iterations each:





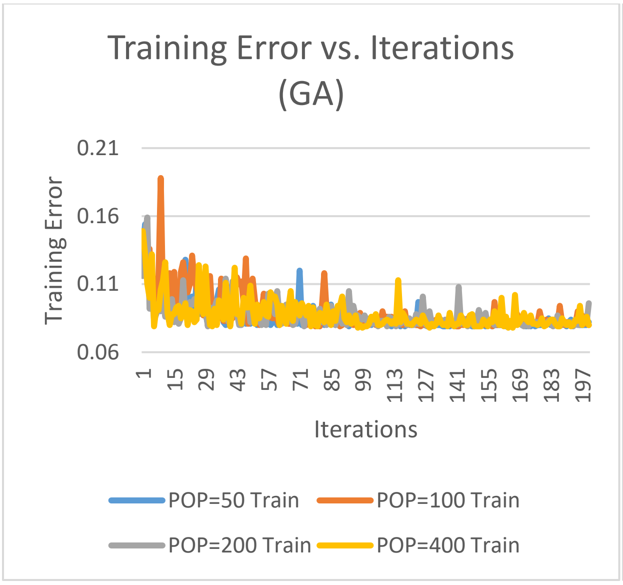
We immediately notice that the value of the starting temperature seems to have a near-negligible effect on the error. However, there is some effect, and we can in fact determine an optimal value for the starting temperature, using similar logic that we used to determine the cooling rate. Note that 1E11 was the default value given for the temperature. Can we improve on that value? We can see that 1E11 actually converges to the optimum somewhat slower than all our other values. On this basis, we'd ideally pick some other value that converges more quickly. However, as in the previous example with cooling rate, we'd also like a value that performs consistently better than the others. What is the best compromise between these two goals? 1E12 seems to be the optimal temperature, as it performs only marginally worse than 1E11 for the first ~150 iterations, quickly surpassing it and reaching the optimum at about the same time as the other temperatures. For this reason, we pick 1E12 as the optimal starting temperature.

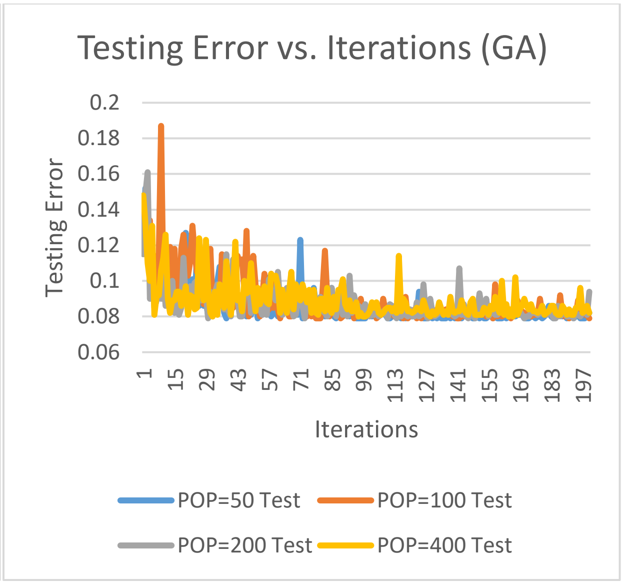
Then, for simulated annealing, we have picked our hyperparameters based off of speed of convergence and consistency of performance across number of iterations. What this has resulted in was a CR of 0.65, and a starting temperature of 1E12, with a final error of 0.079.

1.4 Genetic Algorithms (GA)

Genetic algorithms are, as can be inferred from their name, inspired by the process of natural selection and evolution. Candidate models for the weights of the neural net become individuals in a population, and our error function determines the evolutionary fitness and desirability of traits in those individuals. We then go through a large number of iterations, in this case known as generations, with the models or individuals of a population "mating" (also known as crossover) and producing new individuals with potentially better and better fitness for successive generations. Genetic algorithms also allow us to model mutation, sudden and random changes in the traits of some proportion of the population, that increases genetic diversity, preventing the population of individuals from becoming too homogeneous, and thereby ideally preventing us from getting stuck at a local optimum and suboptimal solution.

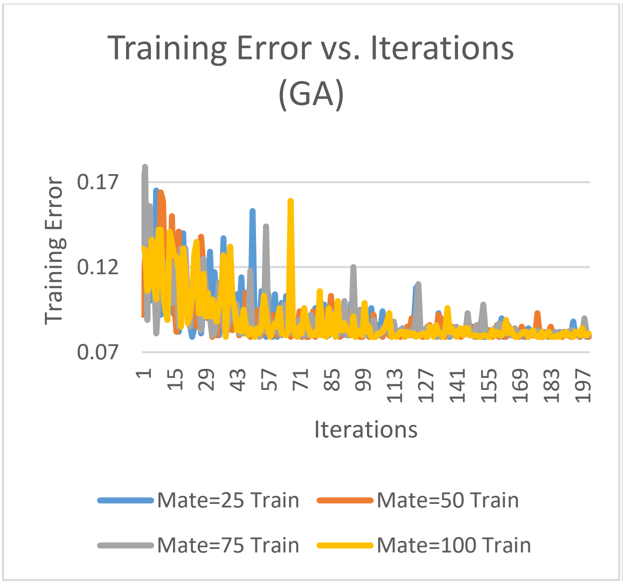
We thus, logically, have three hyperparameters to tune: population size (POP), the number of new individuals with each generation (Mate), and the number of individuals mutating with each generation (Mutate). We begin by observing 4 different population sizes, averaged over 3 trials of 200 iterations each:

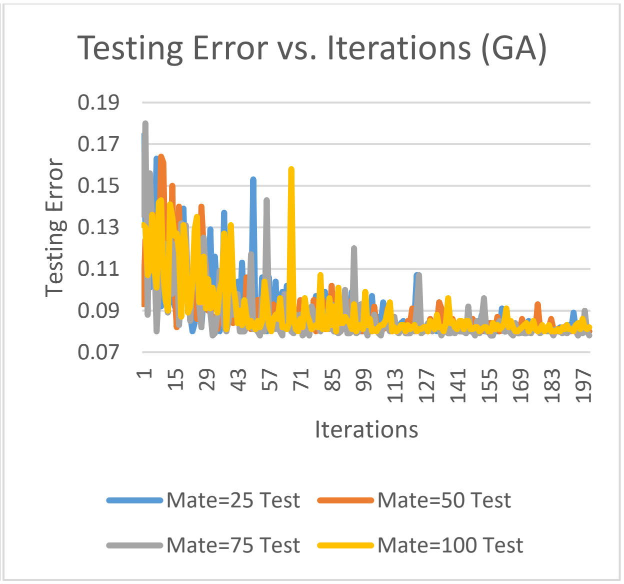




As perhaps can be expected with genetic algorithms, the error seems to behave erratically, only showing a vague trend. Moreover, the population size doesn't seem to have any sort of effect on the error. This would make sense, given that the population size only states how many hypotheses we can have for any given iteration. On this basis, for the sensible values that we picked, this was no handicap at all. However, if we picked very small or very large populations, we would clearly start to see some change in the error: if we had very small populations, we would either very slowly converge to the goal or the population would quickly become homogeneous and get stuck. If we had very large populations, we may reach a point where there would be no variability as the population would be so large it would have all possible individuals; and at this point we wouldn't be able to identify optima, and the algorithm would effectively be a primitive and very slow random search. On this basis, we pick 100 as a safe value for the population, on the basis that it covers enough hypotheses and is not too large to slow down the algorithm.

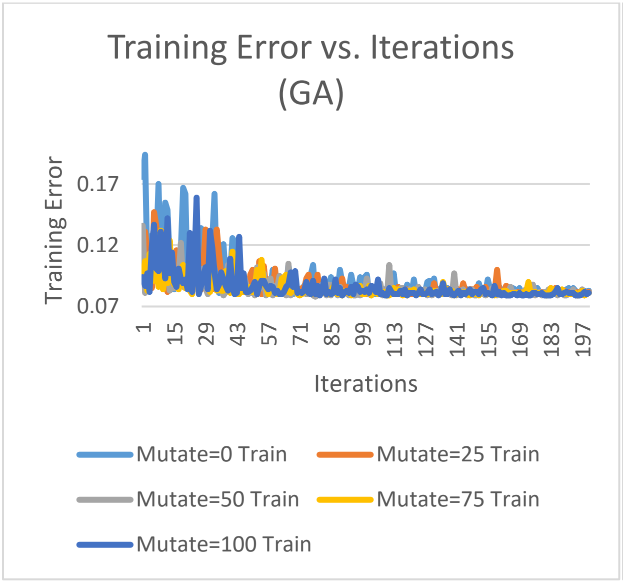
The next hyperparameter to discuss is Mate, the number of new individuals with each generation. For this hyperparameter, we again picked 4 different values for Mate, and tested each of them, averaged over 3 trials of 200 iterations each, with a mutate value of 0:

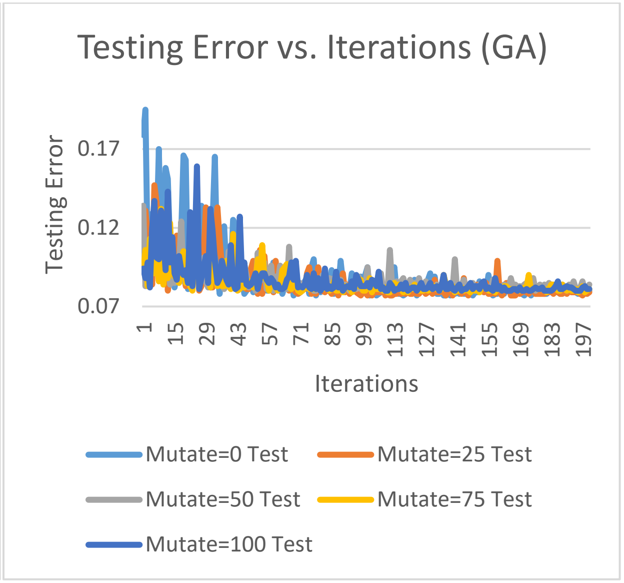




While also looking erratic, given the impact of Mate on the power of the algorithm, in this instance it is imperative we pick a good value, even if by the minutest of differences. We will also note quickly that these experiments were run with a Mutate of 0, resulting in populations that quickly became homogeneous after few iterations (relative to previous algorithms). However, all such values were able to converge to a global optimum, meaning that the population size and mate values were sufficient to some degree; so, which one do we pick? We again want to see some sort of balance between convergence speed and consistency. On this basis, we should pick Mate=50, as it provides the best middle of the road performance, not having as high peaks or as low valleys as the other values.

The final parameter to discuss is Mutate, the number of individuals that mutate with each generation. As such, we picked 5 different values for Mutate, and averaged over 3 trials of 200 iterations, with POP=100 and Mate=50:





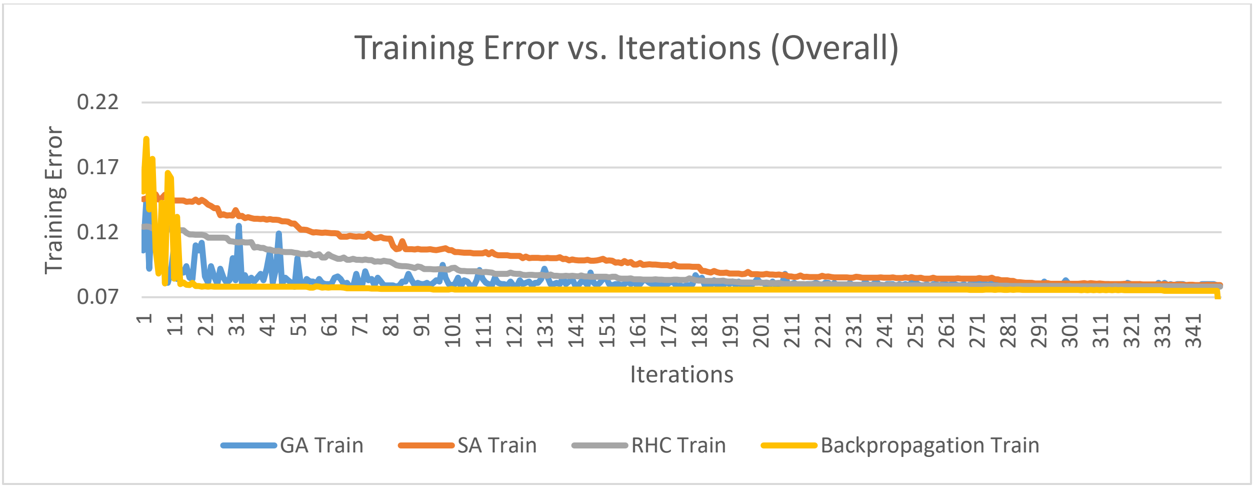
Again, erratic looking curves, but we can clearly distinguish a Mutate value that is better than the others; it is a balanced value that consistently has low error, with few and very small peaks and valleys. This value is Mutate=50. A value this high relative to the population is actually rather unusual. This would imply that the generations need substantial amounts of genetic diversity, proportionally, to remain viable.

To wrap up genetic algorithms, we picked our hyperparameters based off of consistency and minimizing sudden erratic and outlier behavior. Thus, we have picked POP=100, Mate=50, and Mutate=50, with a final error of 0.08.

1.5 Conclusion

To give a satisfying conclusion to solving the problem that this dataset presented, namely whether or not we could successfully predict a prospective coder's gender, take a look at the table below of the hyperparameters we gathered during analysis; it is accompanied by a graph of the optimized algorithms, over 3 trials of 500 iterations:

Randomized Optimization Algorithm	**Hyperparameters**
Randomized Hill Climbing (RHC)	**Number of Restarts:** 1
Simulated Annealing (SA)	**Cooling Rate:** 0.65, **Starting Temperature:** 1E12
Genetic Algorithm (GA)	**Population Size:** 100, **Mate:** 50, **Mutate:** 50



We can see from the above that all the solutions are potentially viable, eventually converging to the same error. But which algorithm to pick? It looks like backpropagation, the default for our neural networks, is in fact the best option, after all, reaching at least the same accuracy as the other algorithms but much more rapidly and consistently. Why might this be the case? The simple reasoning is that backpropagation makes use of gradient descent, allowing it to determine with near-certainty the direction in which to head to find a very good optimum, allowing it to accelerate towards potential optima, instead of randomly sampling points like the other algorithms.

Now that we've compared the performances of our optimized algorithms, you may have noticed that I did not include the graph of the testing error. Why? You may have noticed throughout our analysis that the training graphs and the testing graphs for each of the algorithms we've analyzed look near identical! In fact, oddly enough, this is not new; when we analyzed neural network hyperparameters in our first paper, we noticed a similar pattern. There are only a few reasons this could occur: one is that the model is perfect. However, this answer is incredibly unlikely, given that what we are attempting to model is about people. The other, more likely reason, is that despite our randomization of our dataset, and using filters to split the data into test and training sets, that our dataset is not representative of the real world. In fact, there is quite a bit of evidence to support this: while we have a dataset that only has two output labels, male and female, one such label is horrendously underrepresented in the data. While we expect there to be a split of the data about 50/50 between those two labels, in fact it is more of an 80/20 relationship. This heavy skewing toward one label across the entire dataset implies that randomized splits are simply insufficient for creating a test dataset that could potentially represent data that we haven't seen before. Since the neural network is being plied with data that is heavily skewed toward one label, knowledge of the distribution provides better predictive power than any attribute, perhaps even 80%, seemingly excellent! The neural network then, would have only a few active nodes that are providing the bulk of the output, with a substantial amount of largely unused and weak nodes for attempting to predict the underrepresented female side of the data, making the network unnecessarily complex. The result is a not better than random chance neural network classifier that goes unpunished by our splitting of the dataset, resulting in the same distribution for both.

The second thing you may have noticed is that all the algorithms seem to converge to a global minimum error; none of them really seem to get stuck anywhere: at least not on average. Given enough iterations, they all eventually reach about the same point. While the skewed dataset plays a large part in causing this, it is further exacerbated by the normalization we performed on the data, essentially removing the majority of the local optima that algorithms could get stuck in. Furthermore, the unnecessary complexity of the neural network as a result of the skewed dataset actually may assist in reducing the number of pathological local optima, again making any algorithm more and more likely to converge to the global minimum.

What does all this mean? Ultimately it means that the original problem is flawed in some manner and the data provided is insufficient, making the problem impossible to solve under the current circumstances.

2 Three Interesting Problems

2.1 Introduction

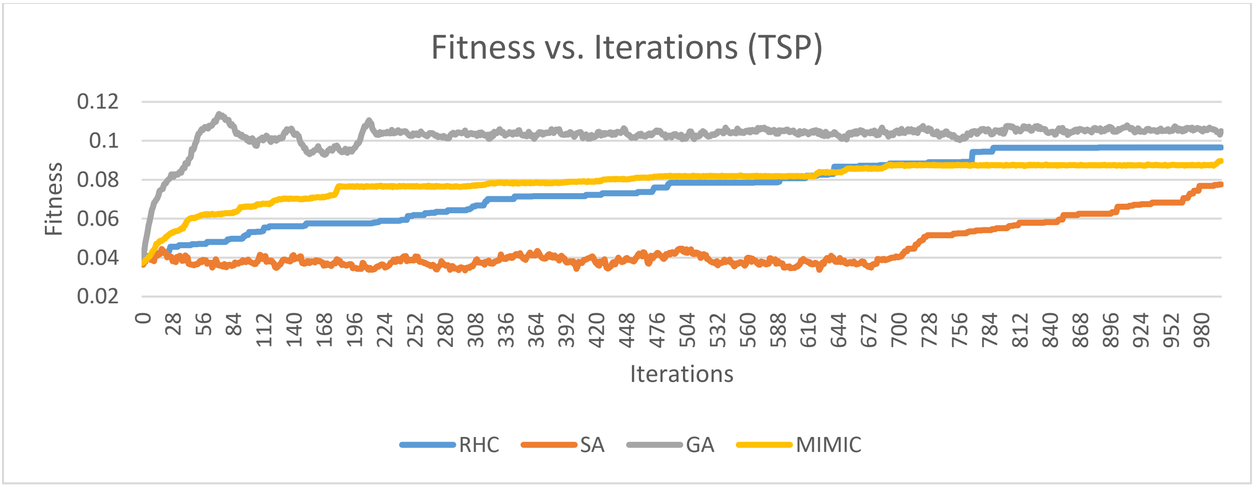
Now that we have worked with a familiar example to analyze the four randomized optimization algorithms, now we can apply what we have learned to some new problems. We will be analyzing the performance of the algorithms on the traveling salesman problem, a simple flip flop bits problem, and finally the max **K**-coloring problem from graph theory. The first will highlight ABAGAIL's genetic algorithm, the second will highlight simulated annealing, and the third and final problem will highlight the MIMIC algorithm.

2.2 Traveling Salesman Problem (GA)

The traveling salesman problem (TSP) is a famous NP-complete combinatorial optimization problem that asks a very general question: given a list of cities, and the distance between each pair of cities, what is the shortest possible path a traveling salesman can take to visit each city exactly once which also takes him back to his city of origin? I tested the four random optimization algorithms we are all familiar with. On randomized graphs of size 50, we ran our algorithms with the following parameters, again found to be optimal by experimentation:

Randomized Optimization Algorithm	**Optimal Parameters**
Randomized Hill Climbing (RHC)	**Iterations:** 20,000
Simulated Annealing (SA)	**Iterations:** 20,000, **CR:** 0.95, **Temp:** 1E12
Genetic Algorithm (GA)	**Iterations:** 1,000, **POP:** 200, **Mate:** 150, **Mutate:** 20
MIMIC	**Iterations:** 1,000, **POP:** 200, **KEEP:** 100

With these parameters, I ran an experiment to determine which tuned algorithm performed the best for the TSP:



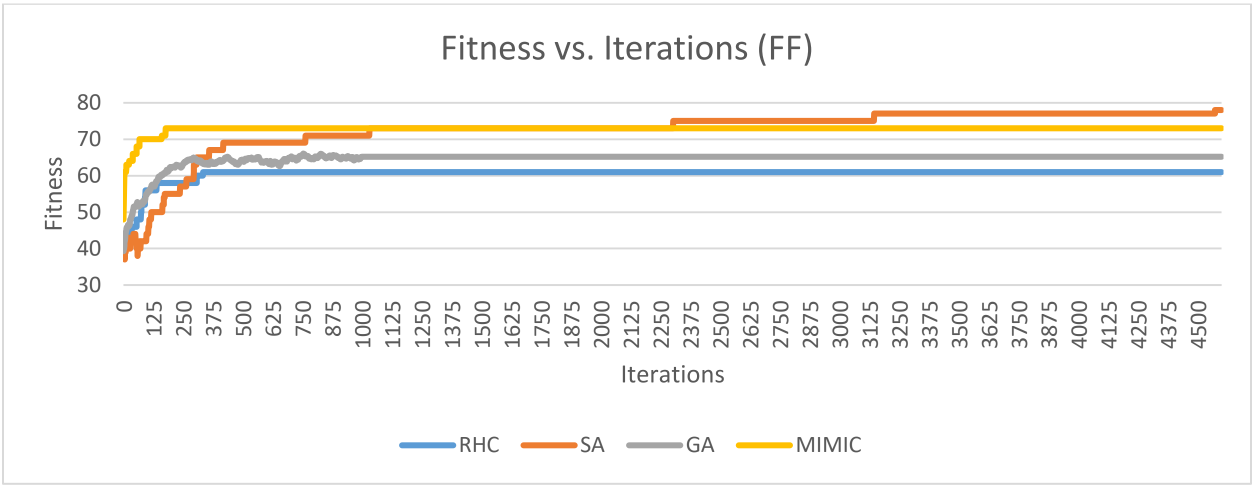
As the graph above clearly indicates, GA performed the best of the four algorithms, resulting in the best fitness overall. It also consistently reaches higher fitness with fewer iterations, with SA and RHC quickly flattening out and getting stuck at optima. However, iterations are not the whole story, as GA took substantially longer to run than SA or RHC, although was still two orders of magnitude faster than MIMIC, while performing better. But why does GA do so well? Seeing as how the TSP is NP-complete, this means that computing exact solutions takes absurd amounts of iterations. However, GA, being nondeterministic and modeled on biological processes, lends itself incredibly well to rapid heuristics, allowing it here to outperform the algorithms in fitness and in number of iterations. Because of this, for practical purposes, GA is the ideal solution.

2.3 Flip Flop (SA)

Unlike the traveling salesman problem, Flip Flop (FF) is a very simple problem indeed; it is a fitness function that takes bit strings as input and returns the number of times that bits alternate in a bitstring. That is to say, going from a number to any other number in the next digit will add 1 to a count, beginning from 0. This count is what is eventually returned by the function. An example is that “1111” would return 0 while “1010” would return 4. A maximum fitness bitstring would be one that when passed into the function, returned its original length; by this logic, the ideal bitstring would consist entirely of alternating digits. I ran all four algorithms, with the following parameters, as originally provided by ABIGAIL, which proved to be the best, I found, and a bitstring length of 80:

Randomized Optimization Algorithm	**Optimal Parameters**
Randomized Hill Climbing (RHC)	**Iterations:** 20,000
Simulated Annealing (SA)	**Iterations:** 20,000, **CR:** 0.95, **Temp:** 100
Genetic Algorithm (GA)	**Iterations:** 1,000, **POP:** 200, **Mate:** 100, **Mutate:** 20
MIMIC	**Iterations:** 1,000, **POP:** 200, **KEEP:** 100

Knowing the optimal parameters, I ran an experiment to determine, for each algorithm, the number of iterations required to reach their optima, as well as the amount of actual time that was taken:



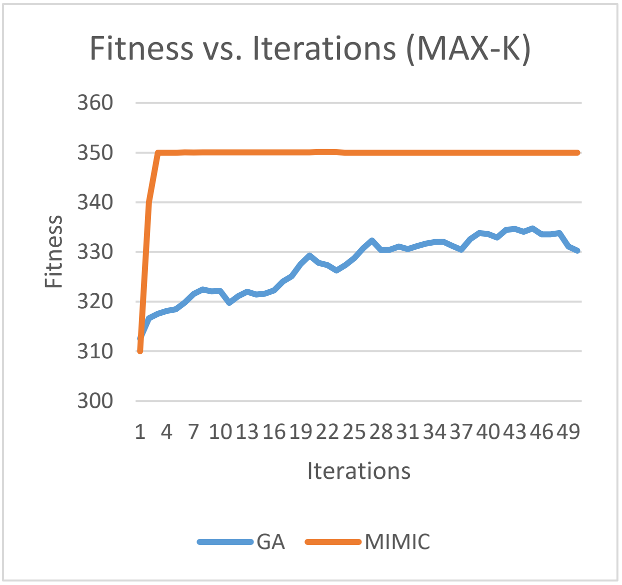
As we can tell from the graphs above, despite simulated annealing taking a substantial amount of iterations, its iterations are several orders of magnitude faster than iterations for GA or MIMIC. This allows SA to reach a better optimum than MIMIC in much shorter time (0.003 seconds vs 1.83 on average!). This optimum is noticeably better than the other two, meaning SA is the winning algorithm overall, performing a better job than MIMIC, the nearest competitor, in time more comparable to the other two algorithms. This makes sense given the nature of SA and the problem at hand. The problem at hand is incredibly simple, with an incredibly fast fitness function; this already makes GA and MIMIC slow and unwieldy, no matter what answer they may give. On the other extreme, RHC is at least as fast as SA, if not faster, but given the nature of bit strings, there are a proportionally high amount of local optima which RHC could get stuck at, giving SA and it’s idea of controlled cooling and temperature a major advantage. For this reason, a higher CR is preferred as well, meaning consideration of more optima by the algorithm. For such reasons, SA is the preferred algorithm for Flip Flop, being the perfect match for the dataset and the simplicity of the problem.

2.4 Max K-Coloring (MIMIC)

To understand this problem, we define a graph as **K**-colorable if each node of the graph can be colored with one of the **K** colors and have no nodes of the same color connected to each other. Much like the traveling salesman problem, the maximum **K**-coloring problem is known to be NP-complete, and is defined as finding some **K**-coloring of a graph that minimizes the number of pairs of identically colored and connected nodes. The graphs used are of size 50, and 8 colors, and for each graph, there exists a solution. The fitness function is based off of whether an algorithm can find the solution, and how quickly the algorithm can find it. I then used the following parameters:

Randomized Optimization Algorithm	**Optimal Parameters**
Randomized Hill Climbing (RHC)	**Iterations:** 20,000
Simulated Annealing (SA)	**Iterations:** 20,000, **CR:** 0.1, **Temp:** 1E12
Genetic Algorithm (GA)	**Iterations:** 1,000, **POP:** 200, **Mate:** 10, **Mutate:** 60
MIMIC	**Iterations:** 1,000, **POP:** 200, **KEEP:** 100

I used these optimized parameters to run an experiment on each algorithm, looking at iterations and fitness again:



The immediate thing to note is that I didn't include RHC or SA; the reasoning for this is that neither algorithm was able to actually find a solution to the problem for any run that I attempted. For this purpose, I am only comparing GA and MIMIC. Building on that, we clearly see that MIMIC rapidly reaches the global optimum far sooner than the genetic algorithm, which must iterate through generations while MIMIC seems to determine the distribution and solve the problem quickly. The question then is to ask why did RHC and SA not find the solution? The issue was that they were taking a lot of data into account, but the key was structure. GA was able to build upon its knowledge of structure generationally, and thus very slowly, while MIMIC deliberately attempted to determine the structure during its iteration, being successful very quickly and thus solving the problem, beating GA not only in iterations but in sheer real time as well, taking only about half of the time in nanoseconds on average.

2.5 Conclusion

As we saw during the first section of this paper, the randomized optimization algorithms that we tried performed poorly compared to the backpropagation we used last time we trained a neural network. I reasoned that this was a result of a highly biased and skewed dataset. That is to say, that the distribution of output labels in our dataset was so heavily skewed that a classifier that randomly chose based off of that distribution would end up with about 80% prediction accuracy, I estimated; although this is also a case of overfitting, since real world data would be nothing like that distribution.

However, despite the failure of randomized optimization algorithms on the dataset I used for my supervised learning paper, that is not to say that they were entirely useless; in fact they proved to be quite useful for solving problems. I proceeded to use them to solve or at least approximate solutions to three interesting problems, one rather simple but with some quirks, and two NP-complete, famously complex problems. That the small array of only four algorithms was able to do that much is an indication that they are indeed quite useful after all.

Moreover, the purpose of this paper was to provide comparative analysis of the randomized optimization algorithms, by throwing them at a variety of problems. We saw where each failed, where each succeeded, and even where they all failed, providing a depth of knowledge about the uses of these algorithms and the problems to which they are most suited.