Yukt Mitash

Professor Hrolenok

CS 4641: Machine Learning

March 4, 2018

Assignment 2: Randomized Optimization

Introduction: In the previous assignment, we examined and evaluated the performance of the following machine learning algorithms: Decision Trees, Neural Networks, Boosted Decision Trees, and k-Nearest Neighbors. We compared their performance across two very different data sets and how that performance was affected by varying the algorithms' hyperparameters. For this assignment, we will be looking at the following algorithms: Randomized Hill Climbing, Simulated Annealing, Genetic Algorithms, and MIMIC. First, we will use one of the data sets from assignment 1 to build a Neural Network. However, instead of using backpropagation to build weights for the neural network, we will use the first three Random Search algorithms to decide upon optimal weights. Then, we will examine three unique optimization problems and compare the performance of each search algorithm on those problems. For the first part of this assignment, we will be using one of the data sets from the previous assignments.

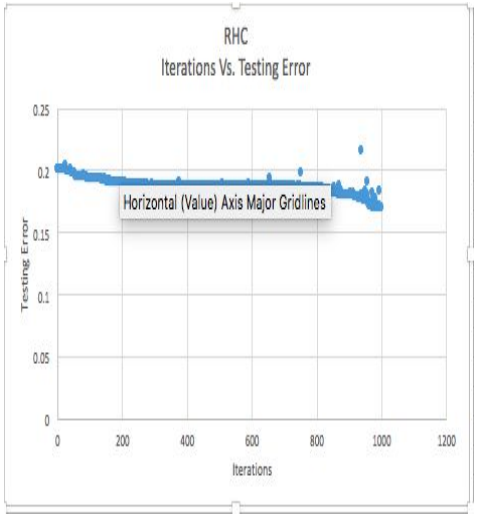
The data set includes the following data on graduate school admissions: *GRE Score, TOEFL Score, University Rating* (rated from 1-5*), SOP* (rated from 1-5), *LOR* (rated from 1-5), *CGPA, Research* (1 or 0 respectively pertaining the whether or not an applicant had done research), and *Chance of Admit* (represented as a decimal). The size of the dataset is 500. To keep the parameters of this problem consistent with the last assignment, the data will be modified as such. The "Chance of Admit" will be represented as a 1 (greater than or equal to 0.725) or a 0 (less than 0.725). This turns the problem into a classification problem that "guesses" whether or not a student will be admitted. Also, we will only be examining the following parameters: CGPA, GRE Score, and University Rating. To learn more about the basis for these parameters, see the previous project here. To learn more about the data set, see the link here.

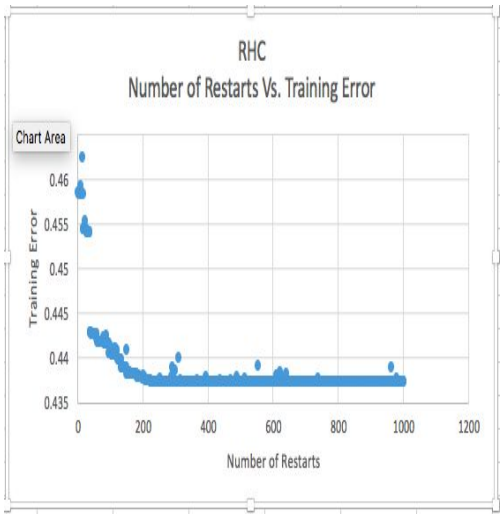
Software: All code is written with the help of the ABAGAIL library. To recreate my experiments, install Java 8 and Ant, clone the project here, use the command line to navigate to src/opt/test/admissionsTesting (this is where I have added all my code) and run the files. Most of the testing files will result in the creation of a .csv file, which can be used to graph the data.

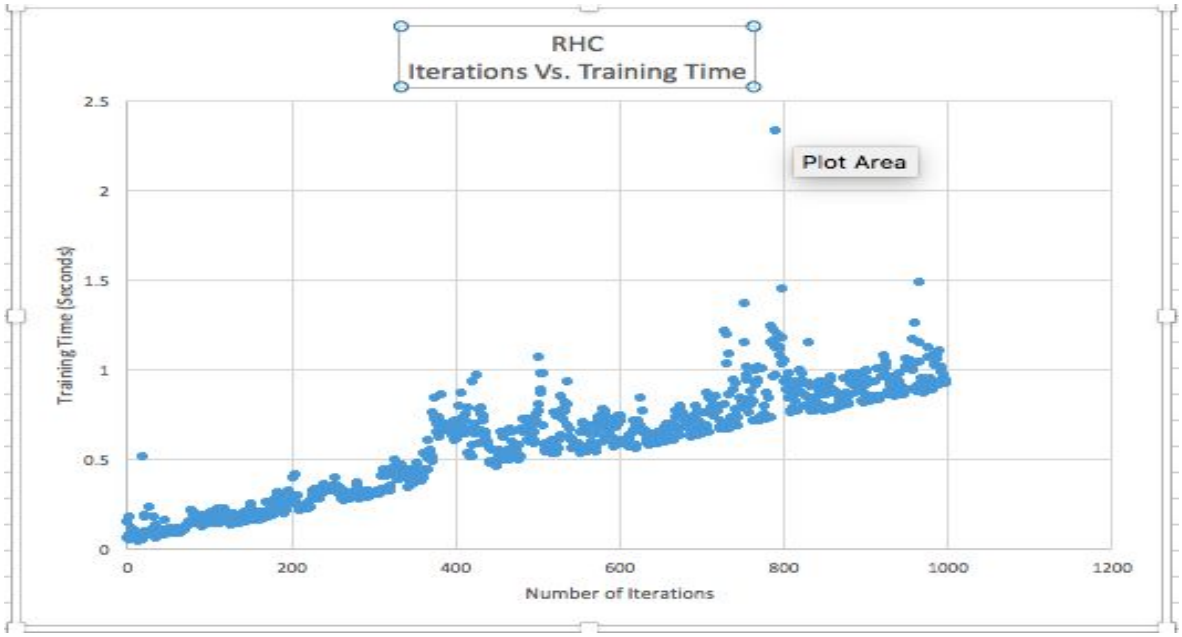
Part 1: Performance Analysis

Randomized Hill Climbing

Overview: Randomized Hill Climbing is a random search algorithm that uses a somewhat "brute force" (relative to other algorithms) technique to find the "best" data in a set. The data that is "best" is defined by a fitness function (how well the data works as a weight for the neural network). Essentially, RHC will randomly pick a point. It will then look at a set of that points "neighbors." Neighbors are defined by another function. If one of the neighbors is better fit, the algorithm will repeat the process for that neighbor. Otherwise, it will save the current point and randomly restart. The purpose of the random restart is to prevent the algorithm from "getting stuck" at a local maxima, believing that the local maxima is the most fit point, when in fact it is the most fit point only within a certain subset of the data. The biggest problem with this algorithm is that even when a large number of restarts are performed, it is still prone to "getting stuck" at local maxima. Given the nature of the algorithm, it is well suited to finding the optimal weights of a neural network. From the tests run in the previous project, I found that the optimal layers of a neural network for this data set is 6. The only hyperparameter that can be varied is the number of restarts, so I examined the performance and training time of the Neural Network when the RHC algorithm used between 1 and 1000 random restarts. Below are some graphs.





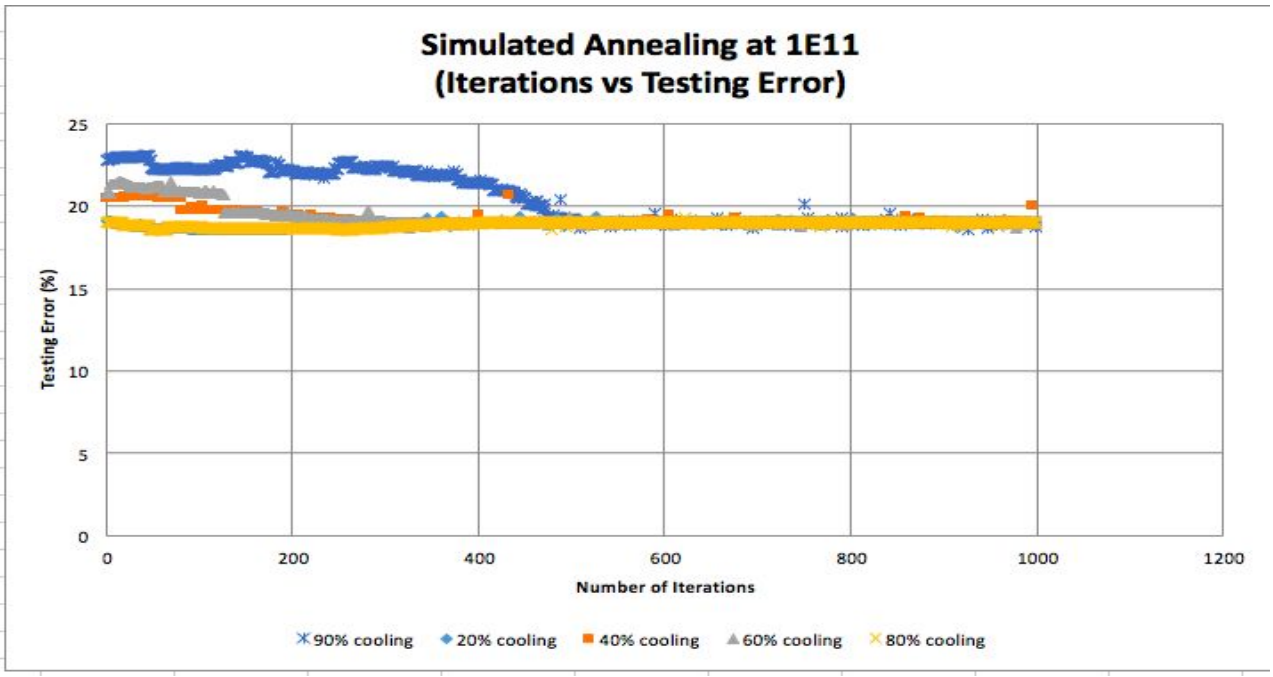


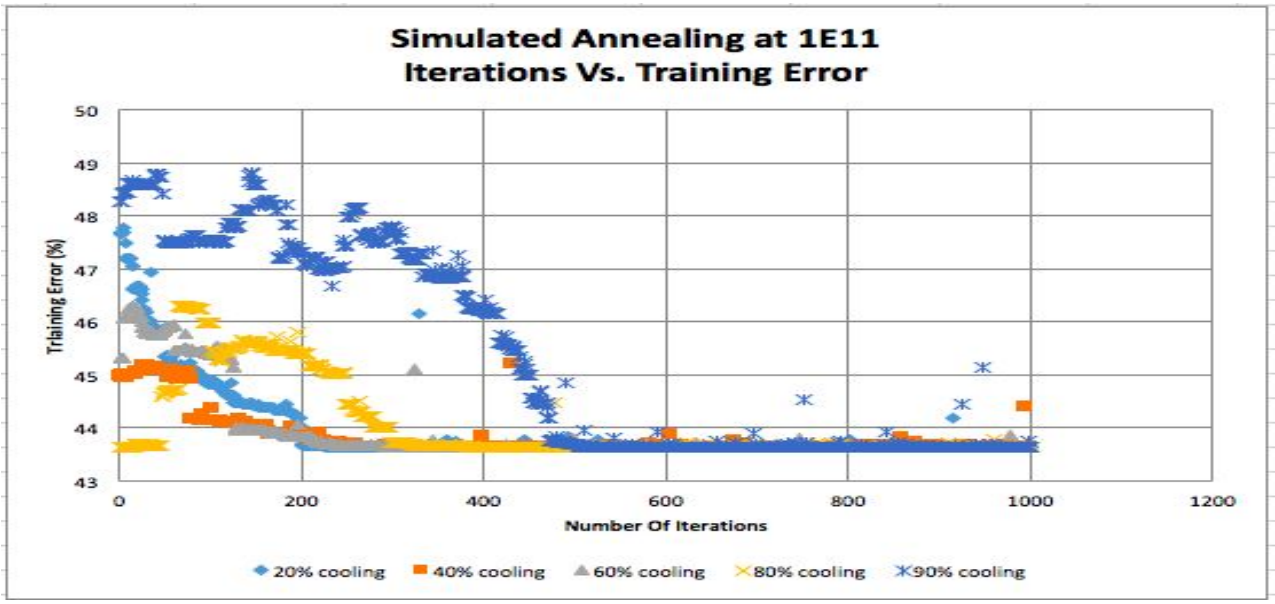
Analysis: Although, there are some issues with the formatting of the graphs above, a couple things are clear. Although the training error converges at around 200 iterations (or restarts), the testing error continues to decrease very gradually until as high as 900 restarts. The reason that it takes so long for the testing error to converge could be because the algorithm is prone to getting stuck at non-optimal weights. This means it would require more and more restarts before the algorithm achieves truly optimal data points. It is also interesting to note that the training error converges at around 0.4375, compared to around 0.17 for the testing data. This is surprising because the algorithm should be better fitted for data it has already seen. This shows that RHC is not at all prone to overfitting. This is because the algorithm is more focused on finding the best possible weights than constantly fitting and refitting weights to specific data. This makes RHC more generalizable. This algorithm outperforms backpropagation for this data set. Backpropagation resulted in a testing score of 0.307 compared to 0.17 with RHC. However, backpropagation resulted in a training error of 0.28, compared to 0.4375 here. This shows that backpropagation is more prone to overfitting than RHC.

Simulated Annealing

Overview: As most readers can likely deduce from the large number of restarts it took for RHC to converge, RHC has a significant problem. It is prone to “getting stuck” at a local maxima, thereby not finding the most optimum point.

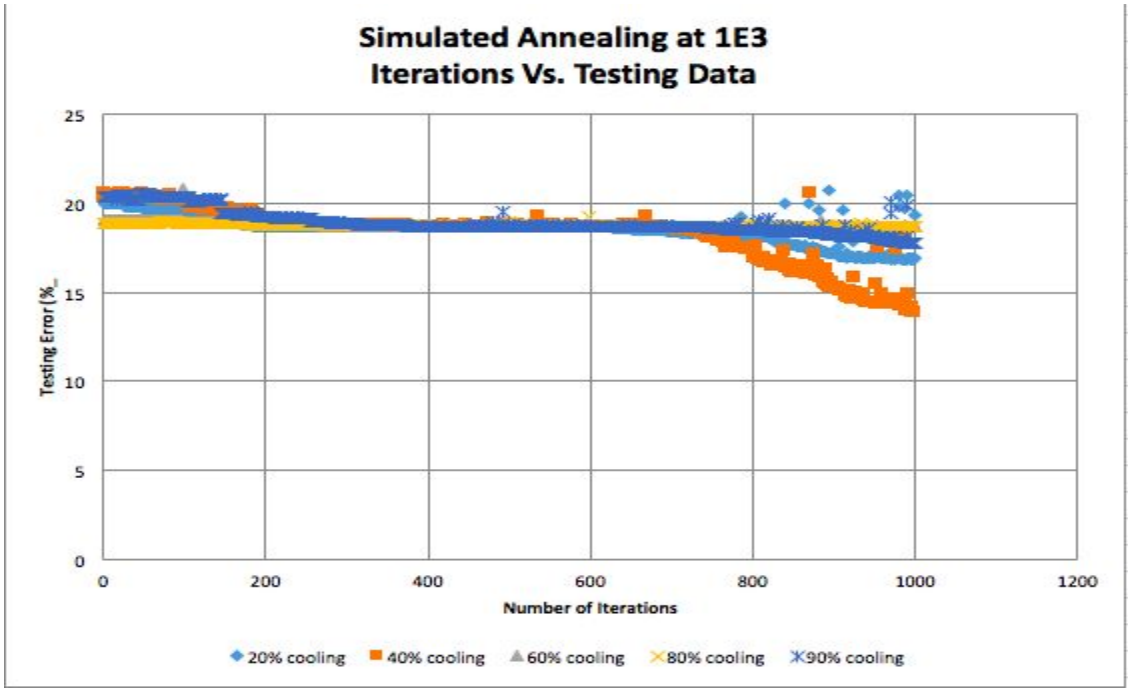
This is an unavoidable side-effect of always looking towards the “best” neighbor. Often times it is better to explore rather than to simply climb. This means moving to points between iterations that are possibly not as optimal as their successor. This makes it less likely that an algorithm ends up at a local maxima. However, we do not want to just randomly move from point to point regardless of fitness. Then we are simply exploring and not climbing. Simulated Annealing solves this problem by simulating the actual process of annealing. Annealing involves heating a metal and then slowly cooling it to strengthen the metal. When the metal is heated the molecules are loosened up and given the opportunity to better fit together. As it cools, the molecules become stronger and more stable, adapting to the structure. The same approach is used with data. The algorithm is given an initial temperature and cooling rate. A new point within a data set is sampled at each iteration. A probability function based on the current temperature and the relative fitness of the new point compared to the current point is used to determine whether or not the algorithm will “jump to” the new point in the next iteration. Early on, when the algorithm is “hotter”, it is more likely to jump to a new point if the new point’s fitness is less than the current point’s fitness. As the algorithm cools down, it becomes more selective about which point to jump to, eventually only jumping to points with a higher fitness. Much like a metal, when the algorithm is hotter it is more active and when it is cooler it is somewhat lethargic. This allows the algorithm to both explore and exploit the data. For this algorithm, the hyperparameters we are going to change are initial temperature and cooling rate. We will use the number of layers determined through testing in Assignment 1. Let’s begin with five different cooling rates and the randomness (heat) at 1E11.

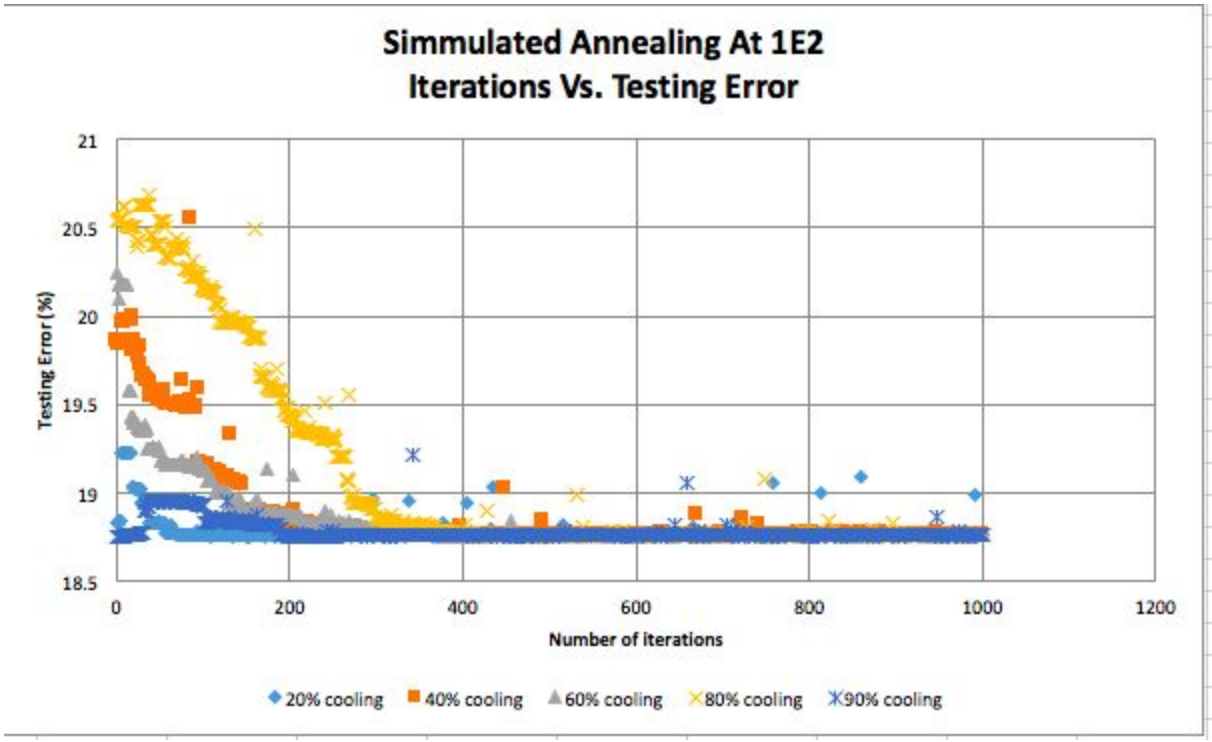




Analysis: It is clear from the above graphs that for all cooling rates (20%, 40%, 60%, 80%, 90%), both the training error and testing error eventually converge at 44% and 19% eventually. This is bound to happen for such a high number of iterations and a relatively small data set. However it is important to note at what point each variation of the algorithm converges. For the testing error, 90% converges around 500 iterations, 80% at around 400 iterations (it is somewhat unclear), and the other three converge around 250-300. The graph for training error follows a similar pattern, but all algorithm variations converge at a much higher percentage relative to the testing error. This shows the similarity between Simulated Annealing and Random Hill Climbing. This algorithm, too, is prone to non-overfitting. It looks for an optimal set of weights rather than constantly refitting the weights to specific data. This causes the algorithm to create very general results and avoid the problem of overfitting.

Essentially, algorithm variations with higher cooling rates take longer to converge. This seemed perplexing to me because it makes intuitive sense that algorithms with higher cooling rates cool faster and thereby converge faster rather than slower. I looked through the ABAGAIL code for Simulated Annealing and found my answer. Essentially, the term “cooling rate” is misleading in the context of the ABAGAIL library. The Simulated Annealing train method uses the cooling rate as a weight to multiply the temperature by at each iteration. This means that, for the ABAGAIL library, a higher cooling rate actually means the opposite! This is because a higher cooling rate means the temperature will decrease at a lower rate. Now that this is straightened out, from here on out, when I say “higher cooling rate,” I mean lower percentage in terms of the above graph and vice versa. Essentially, a lower cooling rate means that the algorithm will be in a state of randomness and exploration for a longer time. It will therefore take longer (more iterations) to converge to an optimal point because it will be spend more time “searching” for the global maxima. A variation of this algorithm with a high cooling rate, will be in a state of searching for a shorter period of time and will thereby converge faster. However, especially for a data set with many local maxima, this variation will be less accurate as it will be more likely to get stuck at a local maxima, missing the global maxima. Now let us look at how the data appears with a higher starting temperature. I will omit the training error because it simply mimics the testing error for this problem.



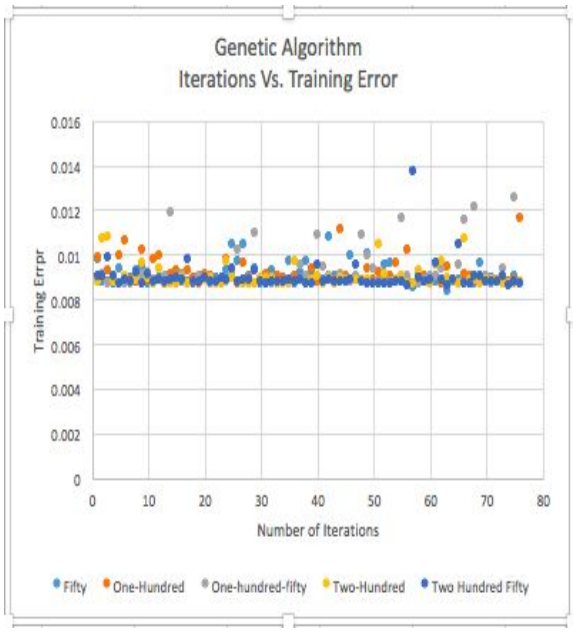


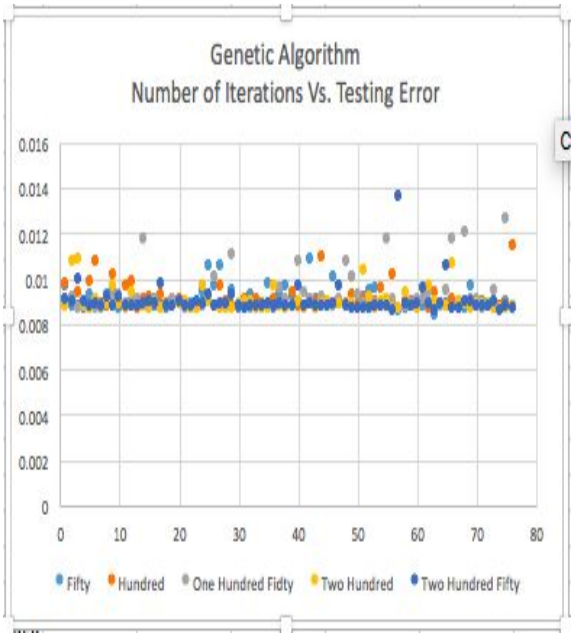
Analysis: In both algorithm variations with lower starting temperatures, the algorithms converges much faster. For a starting temperature of 1E3, the fastest converging algorithm converged at just above 200 iterations. In the case of the 1E2 starting temperature, the fastest algorithm converged at around 100 iterations. This is not surprising. A lower temperature means that less time is spent searching the data and more time is spent “exploiting” the data. In this case, drastically low temperatures do not seem to affect the accuracy of the algorithm. This is likely due to the nature of the data set. If the data set were larger, it would have many local maxima, causing a low starting temperature to

make the algorithm “get stuck” at a local maxima much like RHC. This lower temperature seems to help the accuracy slightly for some of the cooling variations. Perhaps this lower temperature is simply the optimal configuration for a data set of this size. A lower temperature would emphasize exploiting over searching which is generally better for smaller data sets, hence the slightly higher accuracy.

Genetic Algorithms

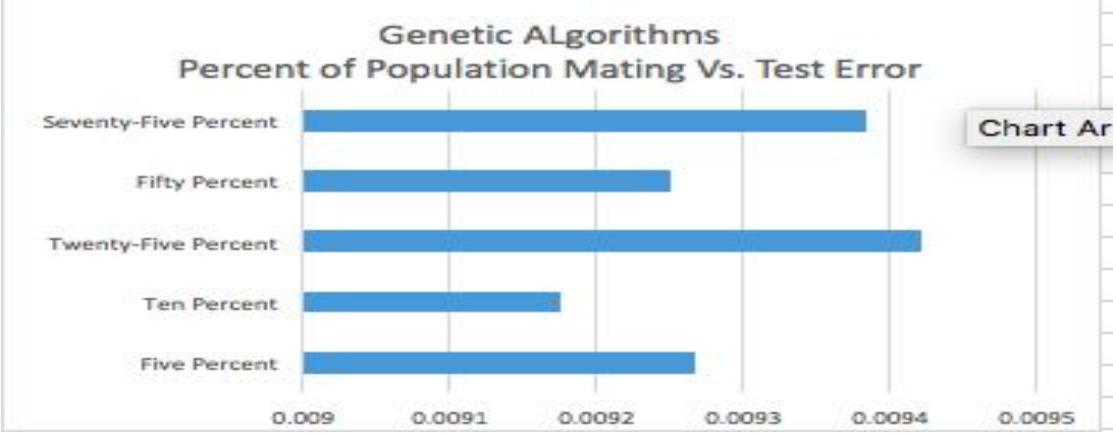
Overview: We will also use genetic algorithms to find optimal weights for our neural network. The nature of genetic algorithms proves them to be quite helpful for this particular problem. Genetic algorithms are modeled after evolutionary biology. The set of data is referred to as a population. The idea from RHC and SA of local search in which an input data point is changed is called a mutation. There is also the idea of crossover where features of two data are combined to create a more fit datam. Finally, each iteration is referred to as a generation. The basic process for a Genetic Algorithm is as follows. First, the algorithm determines the fitness of all individuals in a population based on a fitness function (in this case, how well each point functions as a weight for the Neural Network). Then the n fittest individuals are chosen. Crossover, the process by which aspects of each datum is combined to create a new point, is performed and the individuals who were not chosen to “mate” are replaced with the offspring of the “fittest” individuals. This continues until the data converges to a certain level of fitness. For this problem the hyperparameters that can be varied are the population size (the number of data we examine at each iteration), the mutations (the number of individuals that mutate each iteration), and the number of individuals that we allow to mate. We will be varying the population and the number of individuals that “mate.” Below are scatter plots that compare performance and iterations for different population sizes.





Analysis: The above graphs show Iterations vs. Training Error and Iterations vs Testing Error for the following population sizes: 50, 100, 150, 200, 250. The graphs reveal some major characteristics of the algorithm. First, population size has no effect on performance. The invisible best-fit curve converges at exactly the same point (around 0.09) for all population sizes for both training error and testing error. The curves also seem to converge at around the same number of iterations (less than 10). It makes intuitive sense that population would not affect the performance of the algorithm, especially for a finding weights for a homogeneous data set as such. At each generation, a population is randomly sampled from all the data. This means that the fitness of the population sampled at each generation should be representative of the entire data set and therefore the size of the population at each

generation should not have a large effect on accuracy. Another notable characteristic is that the data converges very early on (as low as 1 iteration). This is because, at each iteration, the data is mutated until it converges. This means that the performance will not vary by that much from iteration to iteration. It is also notable that the testing error and training error have almost identical graphs, meaning that Genetic Algorithms, for this data set, behave the same on data that they have seen and on data that they have yet to see. This could likely be due to the homogeneous nature of the current set. Finally, the reduction in error compared to back propagation is marvelous. The optimal testing error Neural Networks with backpropagation was 0.307, compared to 0.09 here, a 76% reduction. A likely reason for this, is that the backpropagation algorithm is likely to result in overfitting because it is constantly modifying the Neural Network's weight to the training data. This algorithm, on the other hand, is not prone to overfitting because it behaves identically on training and testing data. Now let's examine how changing the percent of the population used for crossover at each generation affects the test error.



Analysis: For this test, I varied the percent of the population that is used for cross over at each generation for a population of 200. I used the following number of individuals for mating: 10, 20, 50, 100, 150, giving percentages of 5%, 10%, 25%, 50%, 75%. I did 77 iterations for each percentage and averaged the testing error to create the above bar graph. I did not include the graph of training data because it was identical. There appears to be very little correlation between the percentage of the individuals use for "mating" and performance. All tests produced roughly the same test error, averaging around 0.0092. This is likely due to the nature of the algorithm. The algorithm is designed to continue mutating the population until the data converges to a single point. It appears that regardless of what percentage of the population is used for crossover, the algorithm will converge at around the same point. Unfortunately the data does not show the rate at which the algorithm converges relative to the percentage of the population used for "mating." This is something that can be studied in future tests by comparing training times of each variation of the algorithm. The data does show, that much like Simulated Annealing, regardless of hyperparameters, GAs will converge at around the same point. It is notable that no matter what the hyperparameters are, Genetic Algorithms are much more useful for determining weights for a Neural Network than backpropagation.

Conclusion: For part 1, we examined how optimally RHC, SA, and GA can be used to determine the weights of a neural network by implementing the algorithms and testing the Neural Network's performance on one of the data sets from Assignment 1. For RHC, we examined how varying the number of restarts affects the training time, training error, and testing error. We found, as intuition would suggest, that training time is directly proportional to the number of restarts. We also found that both the training error and testing error will gradually decrease as the number of restarts increase. Testing error does not converge, but training error does. This makes intuitive sense as increasing restarts increases the likelihood of finding the absolute maxima. The testing error converged at around 0.17, a vast improvement from the 0.307 of backpropagation. For SA, we examined how changing the cooling rate and starting temperature affects the algorithms performance. We found that algorithm variations with the same starting temperature will generally converge around the same point, but higher cooling rates always cause faster converging (fewer iterations). This is important when it comes to saving time. At its optimal level, SA gave us a testing error of 0.14, narrowly outdoing RHC's 0.17 and greatly outdoing the 0.307 of backpropagation. Finally, GA produced the

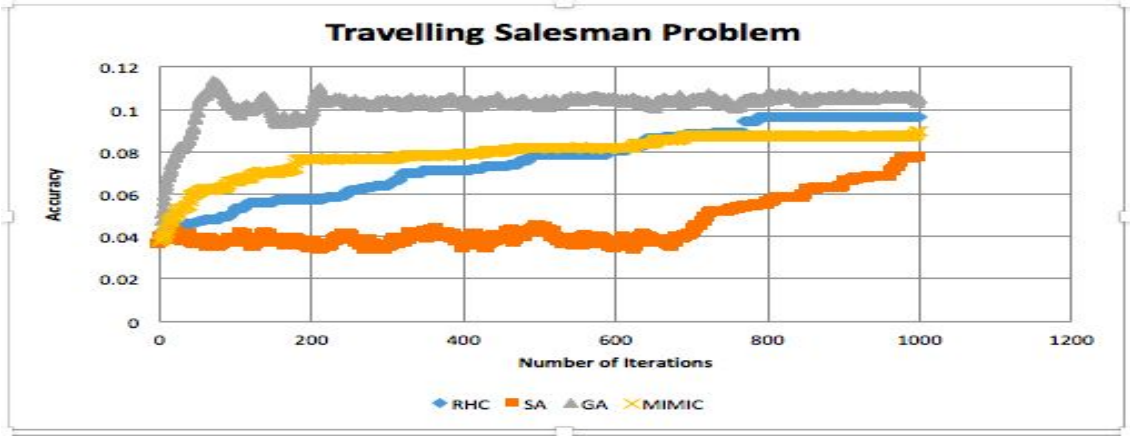
best test results on the Neural Network, averaging a testing error of 0.00915 at its optimal hyperparameters! We found that changing the parameters does not affect the algorithm's performance by much because, the algorithm, by design continues changing the data until it converges. Regardless of the hyperparameters, the algorithm will converge around the same point. Further testing can be done to figure out training times to better understand how hyperparameters affect the rate of convergence. GA outperformed all of the algorithms especially backpropagation. This is likely due to its non-deterministic nature of constantly modifying the data that makes it well-suited to the problem of deciding weights for a Neural Network. One thing that all these Random Search algorithms share in their Neural network results is a lack of overfitting (this occurred with backpropagation as evidenced by its higher training accuracy in comparison to testing accuracy). For SA and RHC the training error was higher than the testing error and for GA the two were roughly the same. This is likely due to the random nature of the algorithms that prevent them from “believing” their data too much.

Part Two: Problem Solving

Introduction: Now that we have analyzed the performance of RHC, SA, and GA for finding optimal weights, we will exam their performance (using the optimal hyperparameters from part 1) in a different problem-space. For this problem-space, we will be examining how these algorithms perform with discrete values. I have stolen three problems designed to showcase the strength of each algorithm: The Traveling Salesman Problem (Genetic Algorithms), Knapsack (MIMIC), and NQueens (Simulated Annealing).

The Travelling Salesman Problem

Overview: The Travelling Salesman Problem is a very well-know problem in Computer Science. The premise is that a salesman starts off in one city and must find the least cost path such that he visits all cities exactly once and ends up back where he started. The cities can be represented by a network of nodes and edges with costs to them. For a very large number of cities, Random Search algorithms can be the best way to find a solution. The algorithms will randomly choose a path from the network that meets the problem’s criteria. A fitness function will determine the value of that path (how low the cost is) and the algorithm will seek to maximize the function based on its specifications. Below is the fitness of each algorithm as a function of the number of iterations.

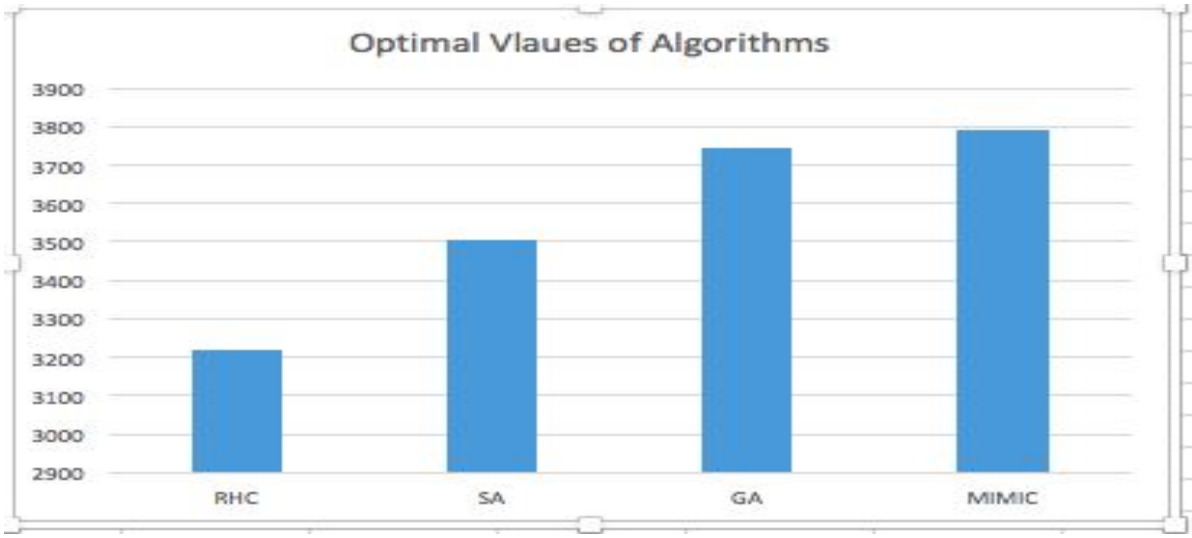


Analysis: The graph shows that GA performed best on this problem and converged with much fewer iterations (as is the nature of GA). At the same time, SA performed worst, even being out done by RHC the very algorithm it is supposed to be optimizing. The reason GA performs so well in this problem space is that the purpose of this problem is to choose an optimal path. Algorithms like RHC and SA will pick one from a large set and examine their neighbors but spend most time searching rather than creating a better path. Essentially, there are many possible paths but only a small number of optimal paths. RHC and SA will spend many iterations examining the suboptimal paths because the nature of those algorithms is to examine neighbors. This wastes a lot of time. GA is designed to create a path from the most fit data. This allows the algorithm to circumvent those suboptimal paths and only look at the best possible paths. This is why SA did especially poor. There are countless suboptimal paths in any data set. With a high

temperature and low cooling rate, SA will spend a lot of time bouncing from suboptimal path to suboptimal path, relying only on luck to find the best path.

The Knapsack Problem

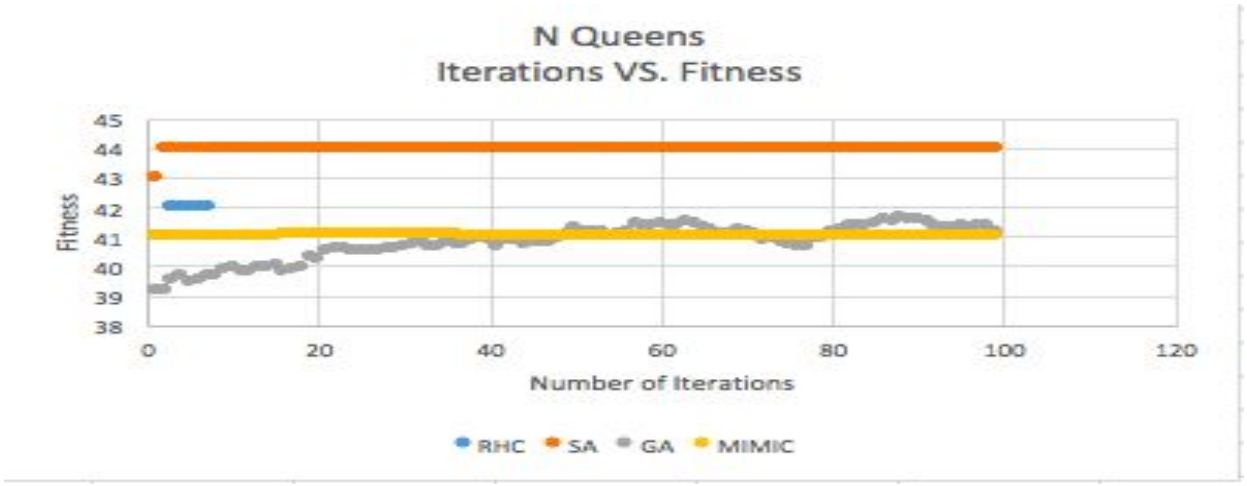
Overview: Another well known optimization problem is the Knapsack Problem. The idea of the problem is that you have a knapsack and a set of items with certain weights and certain values. You want to maximize the value of the knapsack while keeping the sum of the weights at the knapsack weight limit. This problem lends itself well to random search. Each search would start with picking a random set of items that meet the knapsack weight limit. A fitness function would evaluate the data point based on the sum of the values of the weights. The algorithm would then pick the next points based on its specifications. To test all algorithms, I set up an experiment with 40 items that had values between 0 and 50 and weights between 0 and 50. I set up all algorithms with optimal hyperparameters determined from part 1. I tested the algorithms to find the optimal value over 1000 iterations each. Below is the data.



Analysis: It seems that MIMIC created the most optimal solution and RHC created the least optimal solution. The fact that SA created the second-most optimal solution shows that the Random Search technique of simply sampling points is the least optimal for this problem. MIMIC uses a probability distribution to uniformly sample all data points with fitnesses greater than or equal to a certain threshold. The threshold increases at each iteration. AT each iteration the most fit sampled individuals are used to predict the probability function of the next iteration. This algorithm is similar to GA (second most optimal here), except it uses a probability distribution to sample and predict. The function of this problem is finding maximal weights. In a large data set, MIMIC and GA outperform SA and RHC because they can be used to quickly skip over non-optimal points. MIMIC is especially good at this. This allows the algorithm to find the optimal items much quicker.

NQueens

Overview: This problem is based on an n x n chess board with n queens. The queens must be arranged so that no two queens share a row, column, or diagonal. This problem, again, lends itself very well to searching algorithms. Each arrangement can be encoded as a state. Although, this does not have a concrete fitness function, it is clear how the random search would move from one point to its neighbors. The following problem was set up with a board of 10 squares. Here is the fitness of each algorithm as a function of iterations.



Analysis: Although the RHC results are largely covered by SA, it is clear the RHC and SA outperform GA and MIMIC and converge well before GA. This is because the problem is very different from the previous two problems. It does not have such a well defined fitness function that the data can be neatly categorized into points with fitnesses above and below a certain threshold. It does not really allow for the creation of new data through crossover. It does not allow for a probability function to do uniform sampling. The problem does not have the same well-defined degrees of correct as the previous two problems. This problem is well suited towards moving from one state to another by moving one queen around and continuing this process until the goal state is reached. This method is much better achieved through SA/RHC than through GA/MIMIC.

Conclusion: For solving problems with discrete values, no single algorithm is better or worse. It depends entirely on the nature of the problem. If the problem has a well defined fitness function that involves maximizing certain values, MIMIC (or possibly GA) is most likely superior. If the function involves finding a path that is mathematically based with a relatively well defined fitness function GA (or possibly MIMIC) is superior. These two algorithms depend strongly on the fitness function and seek to maximize it at each search. RHC/SA are excellent for cases where one is trying to find an optimal state without a well defined fitness function. If the fitness function is simply whether or not the state is achieved, RHC/SA are superior because they allow for easy transfer from state to state. GA/MIMIC will not perform well in this scenario because they depend strongly on the fitness function. My results could have been improved by creating more graphs examining training times and iterations and studying how performance changes relative to parameters of the problem.

Overall Conclusion: In this assignment we examined the following algorithms: RHC, SA, GA, and MIMIC. We found that for determining the weights of a neural network, GA was by far superior although I could have tested training time more. The reason GA was superior is because there was a well-defined fitness function (how well the neural network performed). As a result, the algorithm was easily able to identify fit and unfit individuals within a population at each generation and perform crossover. However, for solving a certain problem, the optimal algorithm depends on the nature of the problem. Different algorithms perform differently depending on how well defined a problem's measure of fitness is and whether or not there exist degrees to that fitness. For part 1, essentially every algorithm converged at some point. The one algorithm that did not converge was RHC. Towards the thousandth restart, the training error was still decreasing. This is simply due to the fact that RHC is prone to getting stuck at local maxima and thereby not finding the optimal value. This can be solved with more iterations or switching to SA.