Utkarsh Garg
CS 4641 - Machine Learning
Assignment #2
3/14/2015

Randomized Optimization Analysis Report

Letter Recognition Dataset:

The dataset:

This dataset contains labeled examples of letters of the english alphabet, followed by a list of 16 attributes. It contains 20,000 instances which are divided into testing and training sets by the code. There are no missing values. The attributes (all integers) used are defined below:

Horizontal position of box
Vertical position of box
Width of box
Height of box
Total # on pixels
Mean x of on pixels in box
Mean y of on pixels in box
Mean x variance
Mean y variance
Mean x y correlation
Mean of x * x * y
Mean of x * y * y
mean edge count left to right
Correlation of x-edge with y
Mean edge count bottom to top
Correlation of y-edge with x

The number and nature of attributes seems very extensive; the attributes include original readings from the data and then outputs of statistical functions of the same readings.

Attribution: David J. Slate, Odesta Corporation; 1890 Maple Ave; Suite 115, Evanston, IL 60201, from the UCI database.

Why it is interesting:

Having computers identify text from images is a very established machine learning problem with its complexity arising from the fact that there are tons of different fonts and even worse, many different handwritings that make it hard to identify a discernible pattern. Using the supervised learning algorithms that we are testing for this assignment on this problem is interesting because it gives an insight into how much accuracy basic algorithms can achieve on a legacy problem like this.

The problem is interesting from a machine learning point of view because, one the task is interesting, and two because of the attributes provided. Some attributes are spatial attributes on the box itself, while others are spatial on individual pixels. Using these

attributes in the same space can be a challenge, and because no code in this paper tries to deal with the problem, a low accuracy is expected. The data cannot be all treated equally, simply because they do not fit in the same homogenous state space.

Experiment: The Neural net implementation provided in the ABAGAIL code was used. Experiments were run for 10,50,100,250,500 and 100 iterations on the neural net, while keeping the number of layers and number of epochs constant. The iterations were run for randomized hill climbing, simulated annealing and the standard genetic algorithm.

Results:

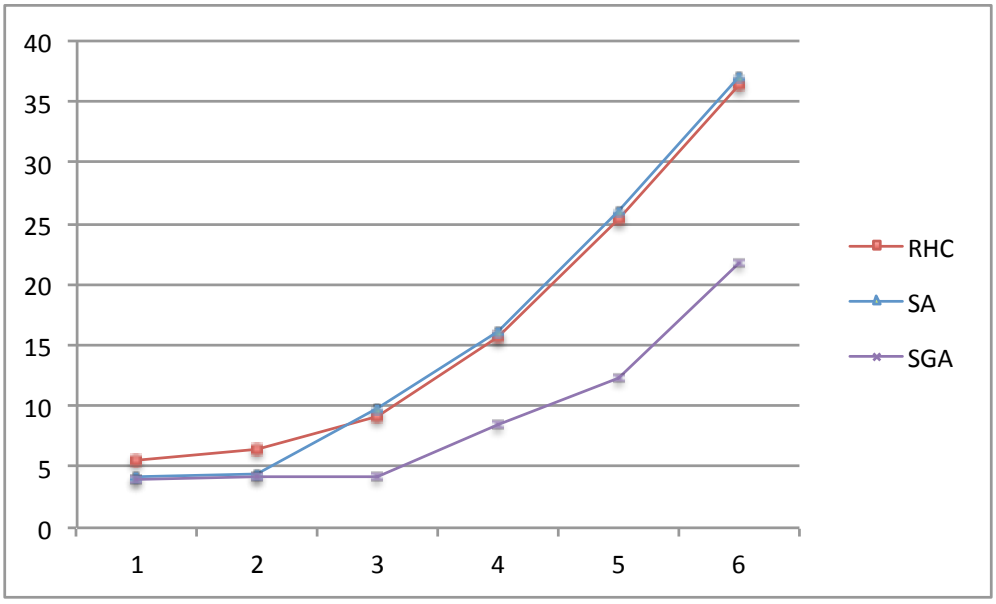


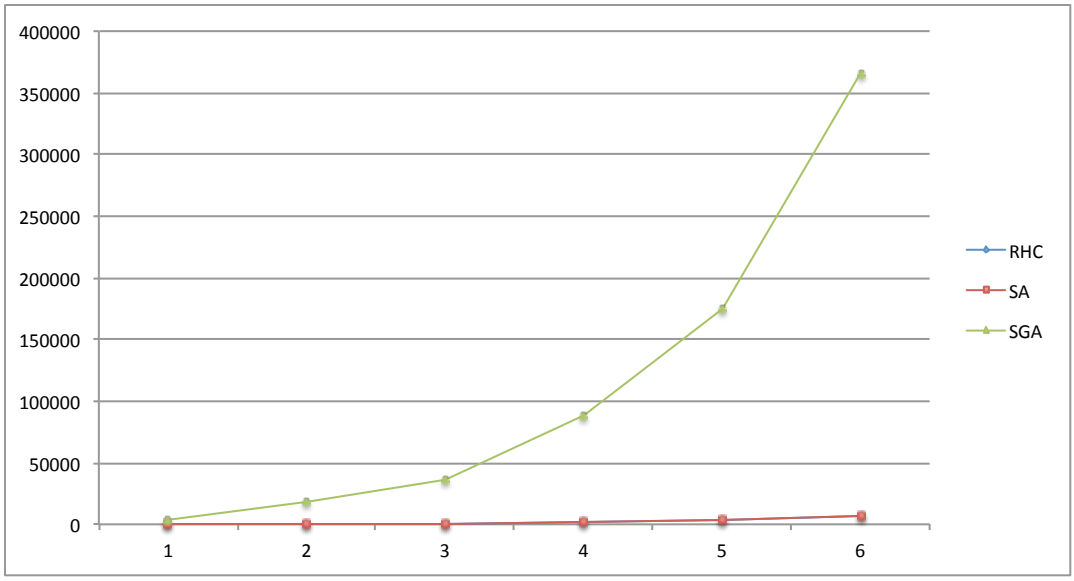
Figure 1. Accuracy% plotted against number of iterations. Note, number of iterations are not from 0-6, but from 10-1000.

Iterations	RHC	SA	SGA
10	5.546	4.114	3.953
50	6.433	4.398	4.273
100	9.182	9.731	4.183
250	15.764	16.146	8.533
500	25.385	26.007	12.281
1000	36.397	37.149	21.677

Table 1. Tabulation of the above result.

The genetic algorithm does not perform well on the dataset. Randomized Hill Climbing and Simulated Annealing fight very closely though. But, zooming out for a moment, none of the algorithms perform well on the dataset, even with 1000 iterations on the neural net. All of them improve in performance, as expected, but none even touches 40% accuracy. I attribute this to the fact that the dataset has 16 attributes and 26 possible outputs. There are more outputs than inputs and that ruins the neural network. This dataset did reach 78% accuracy with back-propagation in Weka, but has reached merely half that value here. This might be because of the way the 2 neural networks are implemented. The neural net in ABAGAIL is not able to see the output as a 4 bit output, but as 26 different outputs. This might cause it to go very to very high dimension functions, thus giving high testing error. Coming back to the comparative performance of the various algorithms. SGA clearly does the worst, and that is probably because of too many possible variations resulting in the same output. The number of mutations might be introducing too much noise. Both, RHC and SA perform much better and are very close to each other in comparison. The dataset is not representative of highs and lows in any way. For each input, it is either of the 26 alphabets. Though there might be a pattern that forms, resembling a hill, assuming uniformly distributed data, I would assume most of the hill would be of the same height and have an equivalent basin. This makes irrelevant, the additional advantage of SA over RHC, thus leading to a very similar performance.

The following is a graph of the training time:



Again, as is expected, the training time of SA and RHC are negligible in comparison to SGA, even though their performance is much better. This simply points to the fact that SGA is useless for this dataset, in this form.

PART 2:

Knapsack: The Knapsack problem is to find the optimal arrangement of objects, each with a weight and a value; optimal is defined as the maximum value while keeping the weight under a certain provided threshold. the two algorithms that consistently performed the bets on this problem are SGA and MIMIC. Knapsack requires balancing weight with values at every step. Both MIMIC and SGA use the concept of populations and work with subsets of the problem. The best known, non random solution of knapsack is through dynamic programming, which solves subsets of problems, culminating to an overall solution. SGA and MIMIC use a similar property, using their concept of populations; they take subsets and solve them. By adjusting the size of these subsets, and by adjusting the randomness of combinations, the output of the algorithms can be significantly modified.

Algorithm results: As is evident from Figure 2.1, the Standard Genetic Algorithm performed the best on the Knapsack problem. Though simulated annealing performed better on some of the iterations, SGA performed the best overall; best here is defined as the highest overall aggregate average over 10 iterations i.e. It found the highest value(best maxima amongst all four algorithms) in majority of the cases (See Table 2.1). It performed on average, 2.31% better thinnest best performing algorithm.

Algorithm	Average
SGA	**4058.137483**
RHC	3899.409778
SA	3930.705271
MIMIC	3966.368955

Table 2.1. Knapsack Average Performance.

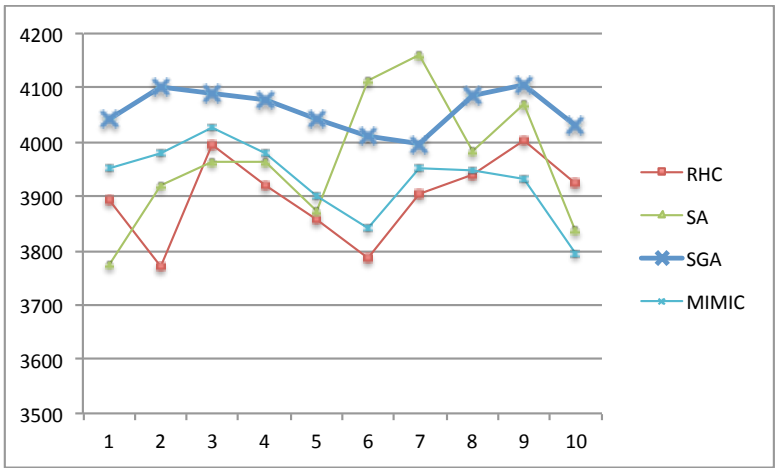


Figure 2.1. Knapsack Performance Over 10 Iterations.

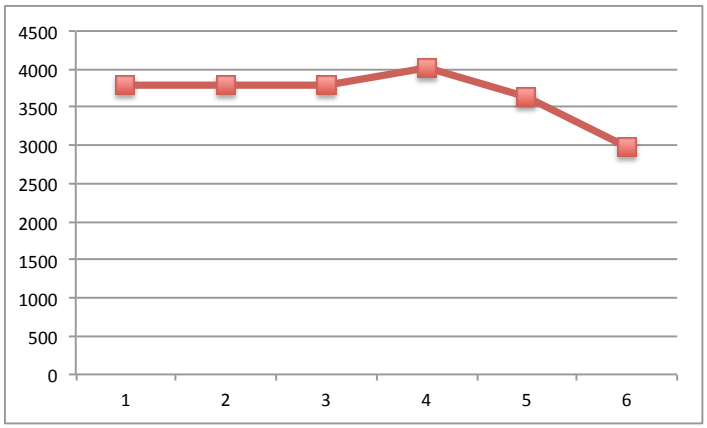
Explanation: The algorithm closest to the best performance (of SGA) was MIMIC. With the same population parameter of 200, and default toKeep(100), toMate(100) and toMutate(10), knapsack always performs better with MIMIC. But, as soon as MIMIC's toKeep is increased slightly, to 150, while the genetic algorithm's mating and mutating parameters are both increased slightly, SGA always performs better. With any increase in MIMIC's parameter, we are directly decreasing the amount of information we gain from it, by eliminating less and less. The default divides the population uniformly and then runs MIMIC, but by dividing into largely uneven sets (150 = 3*50), we are decreasing the total information gain per iteration. In contrast, when we increase the toMate parameter in the genetic algorithm, we increase associations between the data. When we increase the toMutate parameter however, we are directly increasing randomness, it is like introducing noise; some noise is important to avoid overfitting, but beyond a certain point, it simply interferes with the prediction and decreases the result i.e it leads to too many useless/counter-productive mutations.

Experiments with SGA on Knapsack:

Experiment 1 - Increasing the number of mutations: In this experiment, the algorithm's parameters were modified to increase the number of mutations, while keeping the number of matings and the population the same.

Results: As the number of mutations increase beyond 25(the default is 10), the noise increases, and the performance decreases almost linearly. Beyond 25 mutations/ iteration, increasing mutations only create more randomness that the algorithm does not account for.

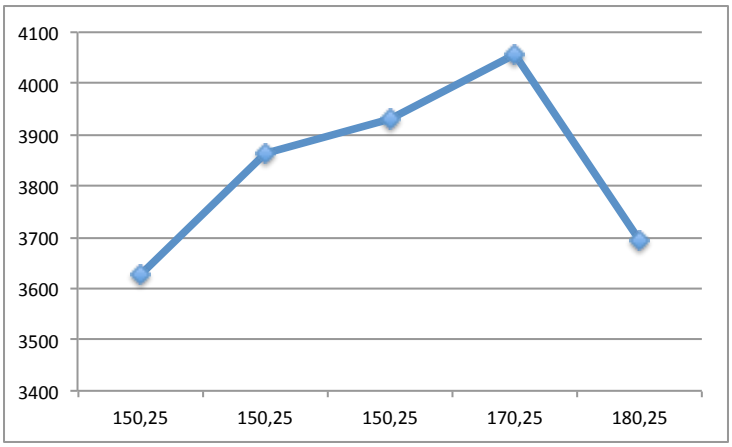
toMate, toMutate	Optimal Value Returned (Over 10 iterations)
150,25	3627.210464
150,25	3862.214429
150,25	3930.705271
170,25	4058.137483
180,25	3693.636637



Experiment 2 - Increasing the number of mates: In this experiment, the algorithm's parameters were modified to increase the number of matings, while keeping the number of mutations and the population the same.

Results: With an increasing number of matings, the algorithm performs better; this is because it is able to form more associations between the data. But, beyond a certain point (170 matings per iteration), the performance drops. I suspect that the reason for this is overfitting, since at this point, every iteration, the algorithm is mating 180 of its 200 population.

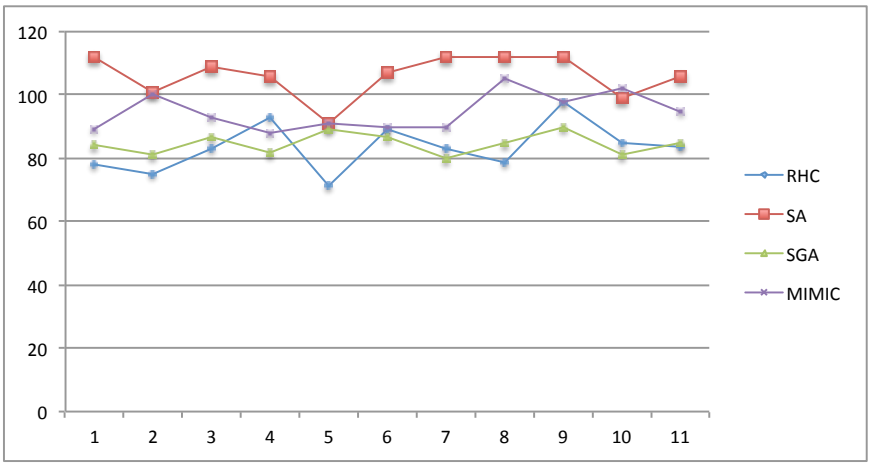
toMate, toMutate	Optimal Value Returned (Over 10 iterations)
150,25	3781.671668
150,25	3781.671668
150,25	3801.773543
150,25	4021.581911
150,60	3649.28705
150,100	2982.038503



Continuous Peaks: The Continuous Peaks problem, as I understand it, looks for the longest substring of 0s or 1s. Like knapsack, it has a limit T, but this is a lower limit that is has to satisfy; continuous peaks tries to find the longest subset of contagious values in a bit string, and earns a reward only if that value is above a certain threshold T. If thought of visually, the entire string of continuous peaks can be thought of hills and valleys, with the hills building up height with continuous strings. Also, this distribution of hills can be very unevenly distributed, with the global maxima covering only a small portion the of the basins, requiring algorithms like RHC to require a lot of restarts and a lot of iterations. This problem is being examined in the context of simulated annealing, which based on a Boltzman distribution, probabilistically jumps from random restarts to random hill climbing. Simulated annealing solves the problem of random hill getting stuck at local optima and of random restart taking to many iterations.

Algorithm results: As is evident from the graph and the table, Simulated Annealing performed the best on the continuous peaks problem, hands down. It performed 12.15% better than the next best performing algorithm, on average. Also, if you look at the graph, there is almost no data point at which SA performs worse than any other algorithm. This graph is over 10 randomly generated iterations. The best is again defined as the maximum aggregate average over all iterations.

Algorithm	Average
SGA	84.6
RHC	83.4
SA	**106.1**
MIMIC	94.6



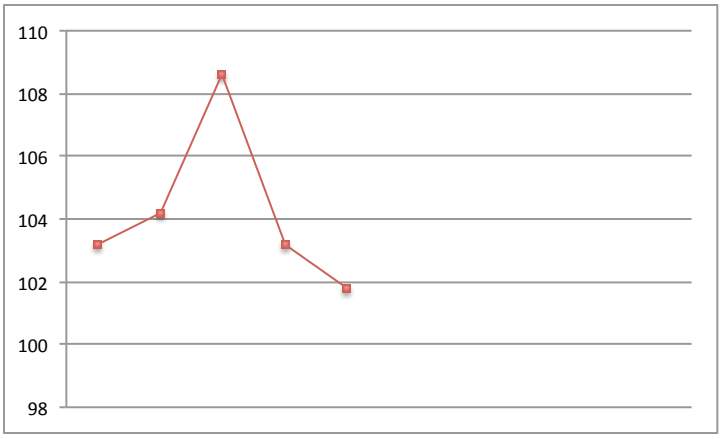
Explanation: Continuous peaks plays on the strengths of simulated annealing. Simulated annealing basically balances explore vs exploit; whether to search for new points and add new domains to the dataset or to optimize and hill climb on known values, and it does this by adjust its temperature variable T. A higher temperature leads to a higher probability of jumping around, thus leading g to more random restarts and thus to more exploration. The cooling exponent parameter of the algorithm defines the cooling rate in terms of number of steps and reduces the temperature. A cool temperature has the algorithm perform like hill climbing as it switches to exploitation/optimization behavior. Simulated annealing segments the space as it explores into domains it can identify will lead to or not lead to a better value/ an optima. with high temperature, it makes broader segmentations, and as the temperature is lowered, it starts ignoring the smaller peaks and always ultimately finds the global optima. This segmentation of the dataset based on temperature is what makes simulated annealing the ideal algorithm for the continuous peaks problem.

Experiments with Simulated Annealing on Continuous Peaks:

Experiment 1 - Decreasing the starting temperature, while keeping the cooling exponent the same: In this experiment, the starting temperature is varied al the way from 1E11 to 1E3, while the cooling exponent remains constant at 0.95. A lower starting temperature means lower randomness and lower exploration.

Results: There is some variability in the results obtained from running the algorithm with a starting temperature of 1E11 (the first three data points), but as the temperature is reduced to 1E7 and then 1E3, the performance drastically drops. Thus, though a higher temperature introduces randomness, it seems like that randomness is necessary for the algorithm to do well i.e. before the algorithm can start optimizing, there is a good amount of the domain space that it needs to explore, to perform well.

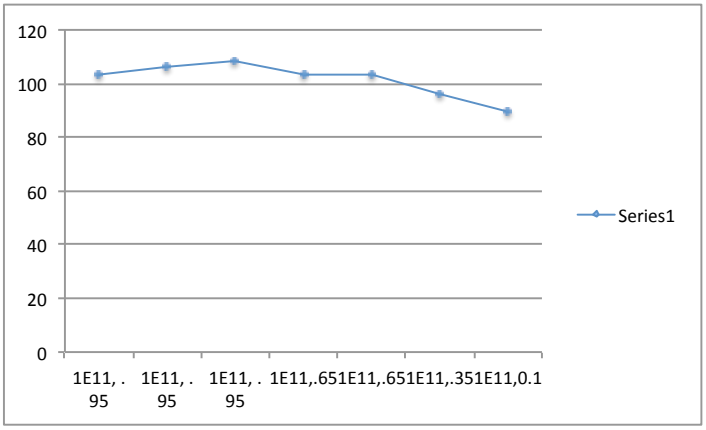
Temperature	Optimal Value Returned over 10 iterations
1E+11	103.2
1E+11	104.2
1E+11	108.6
1E+07	103.2
1E+03	102



Experiment 2 - Decreasing the cooling exponent, while keeping the starting temperature the same: In this experiment, the starting temperature is kept constant, while the cooling exponent remains is varied between 0.95 and 0.1. A lower cooling exponent simply means that the amount of randomness will decrease slower and there will be more exploration in contrast to exploitation for more part of the iteration.

Results: A decrease in the cooling exponent constantly reduces the performance of the algorithm, as can be seen from both the table, and the graph.

Cooling exponent	Optimal Value Returned
0.95	103.2
0.95	106.1
0.95	108.6
0.65	103.7
0.65	104
0.35	96
0.1	89.4

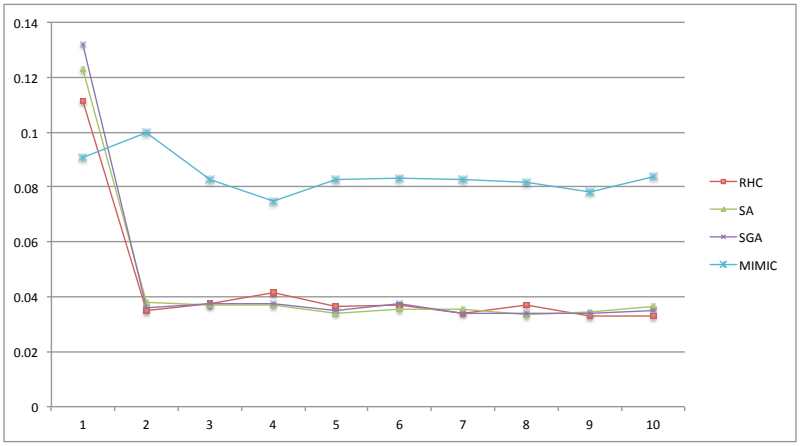


This signifies that a much slower approach to exploitation will decrease the optimality of the outcome. This can be understood by comparing it with random restart hill climbing. Decreasing the cooling exponent means that we approach hill climbing slower, and keep doing random restarts for a larger part of the iteration. This leaves less room for the algorithm to optimize, even though it has explored more space. In context, both the experiments lead to increasing exploration and both lead to poorer performance.

Traveling Salesman Problem: The Traveling Salesman problem operates in a city grid environment. The optimization problem is to find the shortest path (sum of edge lengths) that covers all points cities exactly once. To keep it in context though, we are returning the inverse of the solution, as that is what we are trying maximize.This problem requires a spatial representation of points and cannot be represented simply as a random collection of points, because the length of the edges between points matters, and also, even though two points might be at the same length from another point, they do not necessarily have to represent the same point since they have their own 2D coordinates. This problem is used with the MIMIC algorithm. The MIMIC algorithm is different from other optimization algorithm sin the sense that it introduces structure into the data. It forms dependency trees and using the relations, it updates its probability distribution. Every iteration, it forms a structure i.e. associations between points and then updates its probability model.The traveling salesman problem doesn't just use the arrangement of points in a string, but has many spatial attributes attached to it. This hints at a structural approach.

Algorithm results: Both the graph and the table show MIMIC performing much better than all other algorithms, consistently. MIMIC performs 15.87% better than any other algorithm on average (better - returns higher optimal value). Unlike other problems and algorithms, MIMIC never falls below any other algorithm in performance in this case. The data is over 10 random iterations.

Algorithm	Average
SGA	0.04526629
RHC	0.04360972
SA	0.0444561
MIMIC	**0.0840607**



Explanation: As explained in the description of the problem (TSP) directly caters to the strengths of the MIMIC algorithm. The MIMIC algorithm takes in subsets of points in the forma of a population and then makes associations on them. Then it finds the optimal association and updates the model. This leads to forming optimal subnetworks, a combination of which will culminate to forming MIMIC's guess of the overall optimal solution. If the population parameter is increased too much, the computation gets harder and longer, but the result will be more

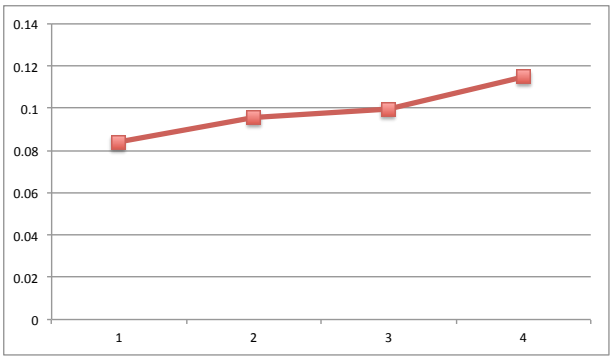
generic, and if the number of data points in each association is increased, the associations might loose some information though as it is not able to form as relevant associations as it might be, with a larger value; increasing the size of one parameters, while decreasing the size of the other should reduce overfitting and ignorance of better associations, though it does become computationally more complex.

Experiments on TSP using MIMIC:

Experiment 1 - Increase the population size while keeping the number of data points for associations constant: This experiment increases the size of each association, while the toKeep variable is kept constant at 10. An infinite size is an NP-Hard problem; thus increasing the size constantly increases computational complexity. The population is varied from 200 to 500.

Results: As the number of data points in the association (population) were increased, performance improved constantly. This is because of the reason explained earlier. It is the attribution to reducing overfitting and giving more global rather than localized results. A localized result in this can be compared to a greedy approach, which is know to perform poorly on the traveling salesman problem.

Population size	Optimal Value Returned
200	0.084086072
200	0.095687992
300	0.099395338
500	0.115168727



Experiment 1 - Increase the toKeep size while keeping the population size constant: This experiment increases the size of the number of datapoint being kept for the association, while keeping the population size constant. The toKeep variable s varied from 100 to 10.

Results: As the number of data points kept for each association were decreased, the accuracy of the algorithm increased. This can be attributed to not being able to extract relevant information because of too much data. It is like saying the training would be done on the entire set, find the relation; it just doesn't work that way.

toKeep	Optimal Value Returned
100	0.047476162
100	0.050543601
10	0.09185945