CS 4641 – Machine Learning
Assignment 2 – Randomized Optimizations

Part 1: Experimenting with the weights of a Neural Net

Dataset

For the first part of this assignment, I decided to use my nursery dataset from the previous assignment about supervised learning. The dataset contains instances of different applications of potential students to nursery schools in Ljubljana, Slovenia. The features such as number of siblings, financial/economical situations of the parents, social status of the families are given in different discrete categories/rankings. For the label, each instance has given a ranking from 1 to 5, ranging from 'not recommended' to 'special priority'. There are total of 12960 instances that have 8 distinct discrete attributes in the dataset. There are no missing fields in the dataset. One important point about this dataset is the distribution of the labels, while three labels have around 4000 instances associated with them; the rest two contains much less instances. This is an import feature about the dataset that needs to be noted when running tests. Even though there isn't a single dominant label, 2 of them seem weak compared to the other 3. However, the dominant labels are found at the both end of the spectrum of rankings, making this dataset still a valuable resource data-wise.

Experiment

After experimenting with ABAGAIL, the machine learning library, I was not able to create an accurate model of a multi-label classification. Instead, I decided to group the labels into two using the domain knowledge I had and since it was kind of a ranking system between 1-5, it made sense. The higher two labels were called 'priority' and 'special priority' applications. So in this experiment, I try to classify an application if it's a 'priority' or 'non-priority' application. In the end, I created a binary classification problem out of the dataset I have used in the previous assignment.

In order to be able to compare different randomized optimization algorithms and their effects on the rearrangement of the weights of a neural network, I re-implemented the neural network from the previous assignment to work with the new binary classification problem I created for this assignment. I used python's *sklearn* library and the generic neural net classifier that comes with it. The default back-propagation algorithm was stochastic gradient descent, which is a very optimized algorithm. Because of that, I was not expecting any of the randomized optimization algorithms to surpass that both in accuracy and runtime.

To see how algorithms behave, I tested those 4 (backprop and 3 randomized optimization algorithms) on a neural net that has 5 hidden layers with 2 neurons on each layer. I compared both the accuracies and runtime values over increasing number of iterations.

Plots

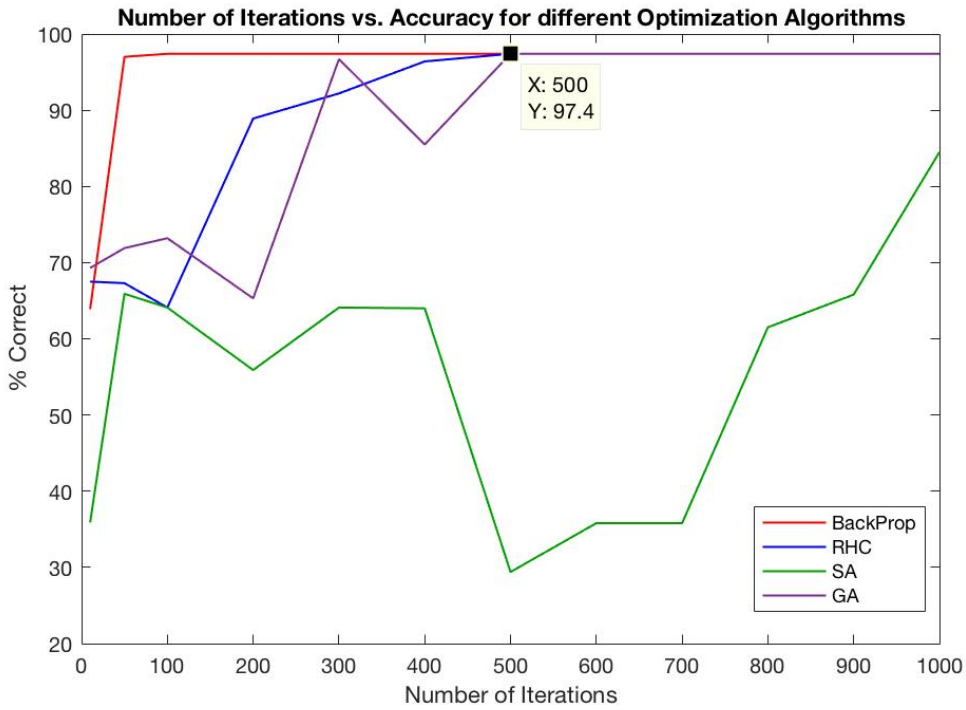

Figure 1: Number of iterations versus testing accuracy on different optimizations for neural network weight change.

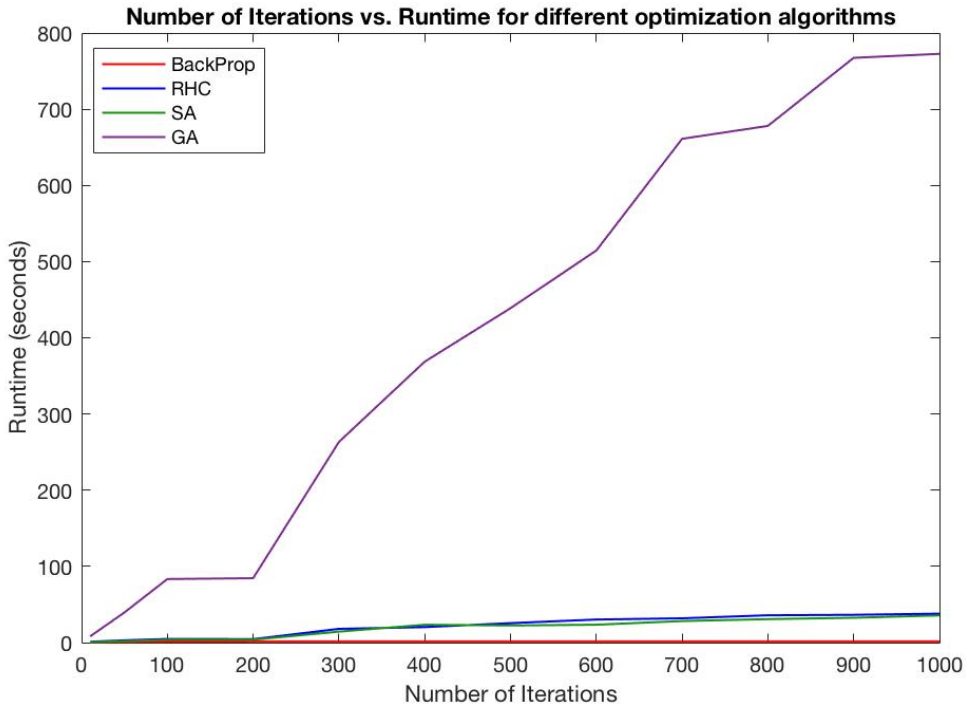

Figure 2: Number of iterations versus runtime of different optimizations for neural network weight change.

Results

The independent variable of this experiment was the increasing number of iterations. I have tested different algorithms on iterations of 10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 and 3 out of 4 algorithms managed to reach to the same accuracy of 97.4% after different number of iterations. As expected, the neural net from the previous homework, the one with the stochastic gradient descent, reached to the maximum accuracy the fastest with around 50 iterations as can be seen in figure 1. On the other hand, simulated annealing performed very poorly compared to the other algorithms and the highest it reached was around 84% with 1000 iterations.

RHC takes around 500 iterations to reach its maximum potential. With the random starts it has, RHC is the simplest algorithm of all. 500 iterations is relatively too much compared to the backprop, suggesting that there are lots of local minima that it gets stuck in.

SA is similar to RHC but it has an improved exploration feature that I will be talking more about in the later sections. However, SA performs the poorest in this case, suggesting that it fails to lower its temperature for the right extreme and exploit more in the global minimum. Looking at the dataset, we see that almost all feature/attribute combinations are covered and a possible equal distribution of labels could be causing more of a flat shaped graph with many shallow local minima, preventing SA to find the global one.

We see that GA also reaches the maximum accuracy of 97.4% with around 500 iterations. The accuracy curve of GA in figure 1 looks similar to RHC's curve. Like RHC, GA choses points randomly and mutates a new population of solutions to choose the best and generate more generations. Taking around 500 iterations, just like RHC does, GA shows us that there are lots of local extremes in the dataset.

Looking at the figure 2, GA obviously take much more time compared to any other algorithms and as expected, backprop takes the least time. The reason for that is manages to converge at the right weight values, causing the algorithm return early when the perfect classifier model is created. The reason GA takes the longest training time is because it generates an entire population for each data point for the number of iterations and this takes up the longest processing time.

Part 2: Random Optimization Problems

Problem 1: Traveling Salesman Problem and Genetic Algorithm

Travelling salesman problem is a well-known problem in computer science and can be solved by dynamic programming. Here we will be using our randomized optimization algorithms to solve the problem and see which algorithm does better. The traveling salesman problem focuses on a graph and basically it asks for a route/path on the graph that visits each of the nodes only once, and comes back to the starting point with the lowest cost possible.

Experiments

For the sake of this homework and to be able to use maxima instead of minima, the traveling salesman problem I was working was trying to maximize the inverse of the total traveling distance instead of minimizing the distance itself. I tested all four algorithms on different sized graphs, ranging the number of nodes from 5 to 100. Each time the code was generating a random graph with N number of nodes and random weighted edges between each node. Then the same graph was being used for all four algorithms for the same N value.

Results

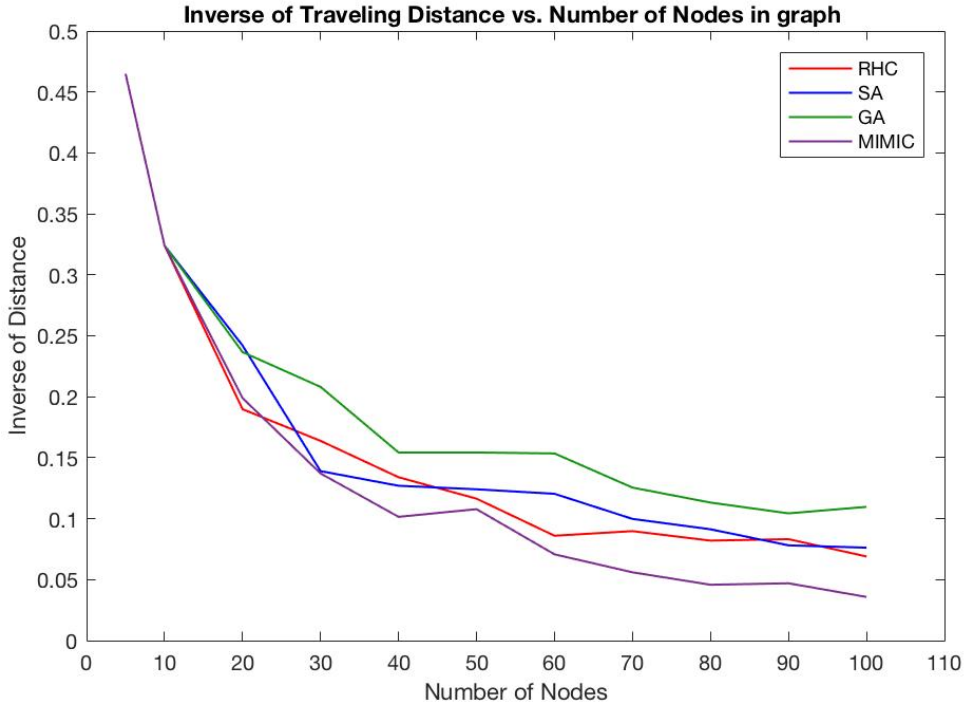

Figure 3: Inverse of the traveling distance versus the number of nodes for the traveling salesman problem

Looking at figure 3, we see the inverse of the traveling distances for diferent number of nodes in a randomly generated graph and how each optimization

algorithm came up with a result relative to each other. First of all, we see that as the number of nodes increase, the inversed distances are decreasing too, suggesting that distances itself are increasing as expected. For N = 5 and 10, all of the algorithms find the same value, suggesting that they all reach to the global maxima. However, after that, we observe that GA is doing better compared to the other algorithms. Throughout the experiment, all four algorithms proportionally find lower fitness scores (distance inversed) and they manage to keep their positions stable relative to other algorithms. This shows that the results of the experiment are not random and there's certainly a difference relationship between the problem and each of the algorithm. The reason GA doing better than others can be explained with the fact that it generates the populations that are similar to each other and in such big a solution space, it doesn't drift apart from the global maxima not as much. However, MIMIC performs really poor because it tries to span the entire solution space and the possibility of finding the right region is just too small. Increasing the sample size of MIMIC would increase its performance, however MIMIC was already taking way too long time compared to the others and doing so would make it even worse.

Problem 2: Flipflop problem and Simulated Annealing

Flipflop evaluation function basically return how many flips happen in a string formed of binary values. It also counts the very first bit (the most significant bit) as a flip since it becomes any value out of nothing (index = -1). So given '0011', the function returns 2 and given '0101', the function returns 4. The maximum number the flipflop function can return is equal to the length of the string given the right input which is an alternating bit string as '1010...101' or '0101...010'.

In the context of an optimization problem, an algorithm would start with a random binary string and try to optimize it by trying to make the string an alternating bit string. However, there are lots of local maxima due to the nature of the problem when an algorithm is flipping only the neighboring bits to optimize the result.

Experiment

I tested different algorithms on this problem and tried to measure their performance on different lengths of bits. If we call the number of bits in a string N, I tested the different algorithms on N values starting from 50 and goes up to 500 in increments of 50. Instead of keeping the iteration numbers same for all algorithms, I tried to keep the training times similar by adjusting the number of iterations because with the same number of iterations, GA and MIMIC took significantly longer times. The results can be seen in the figure 4.

After seeing simulated annealing does better, I also decided to change hyper parameters of SA and see how it acts in different situations. For that, I changed the cooling rate of the algorithm and observed if it works better when it exploring more and exploiting more. After experimenting with very different cooling rates, I decided to run my main experiment with rates between 0.30 and 1.30 with increments of 0.10. The results can be seen in the figure 5.

Results

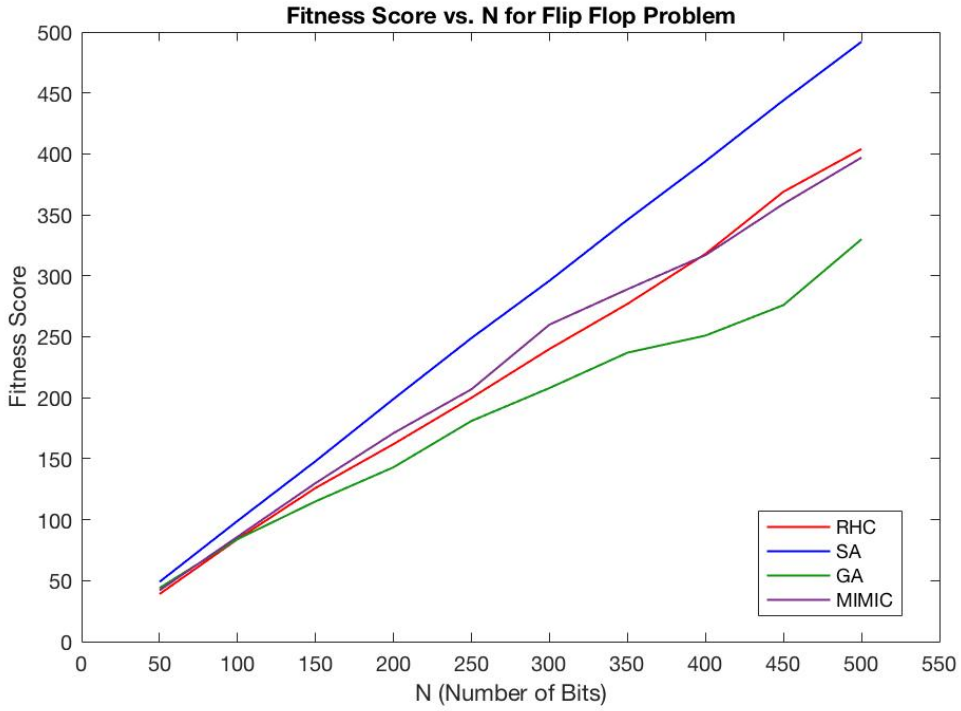

Figure 4 : Fitness Score on increasing number of bits in Flipflop Testing

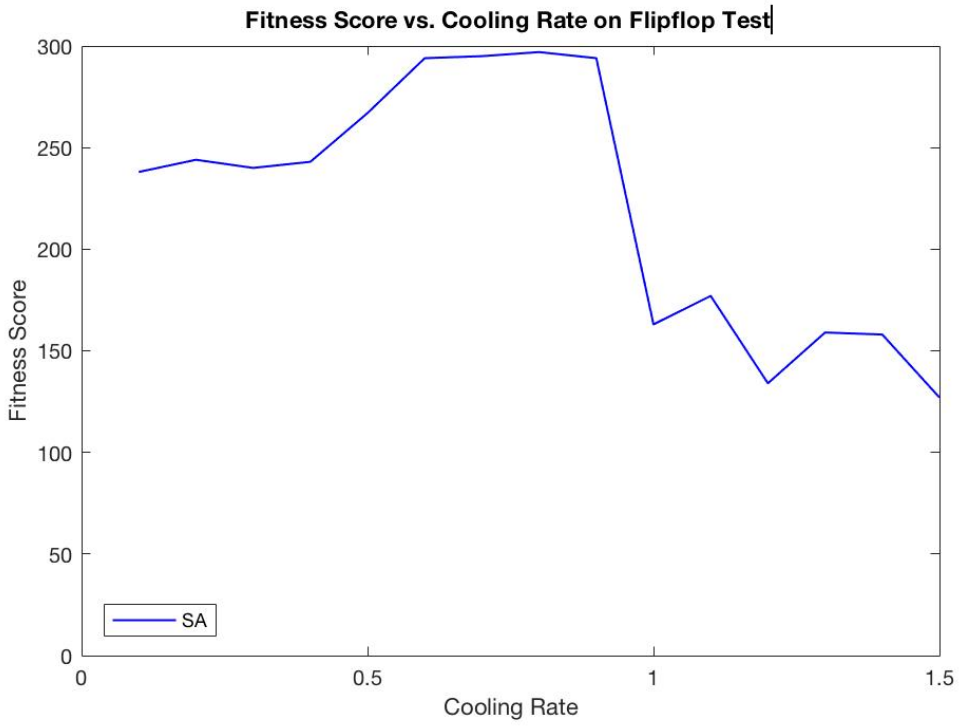

Figure 5: Fitness Score of SA using different cooling rates in Flipflop testing with N=300.

Figure 4 shows how different algorithms perform on the flipflop test with an increasing bit length. It is clear that SA does a perfect job, and it is the clear winner for all N. A simple explanation for this lies beneath the fact that SA works well in a solution space where the local extremes are discrete and close by. When

the peaks are close by, it allows SA to cool off and find the right value. In other words, it exploits more. However, starting with a high exploration rate allows it to find the correct spot.

We see that GA is performing very poorly, that's because the local extremes are very close and similar so generating new population doesn't go beyond being random noise. RHC and MIMIC perform almost the same. However, RHC is only the exploration part of SA, which explains experiment 2 that can be seen in figure 5. As the cooling rate drops, the SA performs worse, suggesting more exploration and less exploitation is not ideal for this kind of optimization problem. It's also clear that only a specific range of values (between 0.5 and 1.0) work as an ideal cooling rate.

Problem 3: Four Peaks problem and MIMIC

Problem

Four Peaks problem is a known optimization problem that has total of 4 peaks, of which 2 are global maxima and 2 are local maxima. Hoe the problem is defined goes like this: Given an N-dimensional input, you're trying to maximize this:

$$f(\vec{X}, T) = \max\left[tail(0, \vec{X}), head(0, \vec{X})\right] + R(\vec{X}, T)$$

Simply, if you're given a string size of N, and trying to get T+1 leading 1's followed by 0's or T+1 trailing 0's preceded by all 1's. However, in such optimization problem, having all 0's or all 1's appear as local maximum since any bit changes that can be made in those states reduce the fitness score.

Experiment

For experimenting, I have tested all four randomized algorithms on this problem, by increasing the number of iterations they need, in order to see if one particular algorithm finds the best fitness score or reaches to the maximum score with the least number of iterations. I used N = 100 and T = 25 for the Four Peaks problem. I also used the following number of iterations: 100, 200, 300, 400, 500, 1000, 1500, 2000, 2500, 3000, 4000, 5000, 7500, 1000, 12500, 15000, 20000. The results of this experiment are given in figure 6 down below.

Another experiment I ran was running the MIMIC algorithm on the same problem (N=100, T=25) again but changing the sample size. I changed the sample size from 30 to 450 in increments of 30. However, I kept the number of sample to keep at 20. My expectation for this experiment was a linear increase in the fitness score that would reach to 100 (max score). The results of this experiment are given in figure 7 down below.

MIMIC

MIMIC is a randomized optimization algorithm and its difference from the basic RHC algorithms comes from its ability to use previously searched data points and

the results of these searches to concentrate special regions of the optimization domain. Overall, MIMIC starts with some uniformly distributed random points on the search space and uses the median of the selected points to update the density estimator that gives the next set of points to choose from. Then it generates more samples and narrows the search space.

Results

Figure 6: Fitness Score vs. Number of iterations on the Four Peaks problem, using different randomized optimization algorithms.

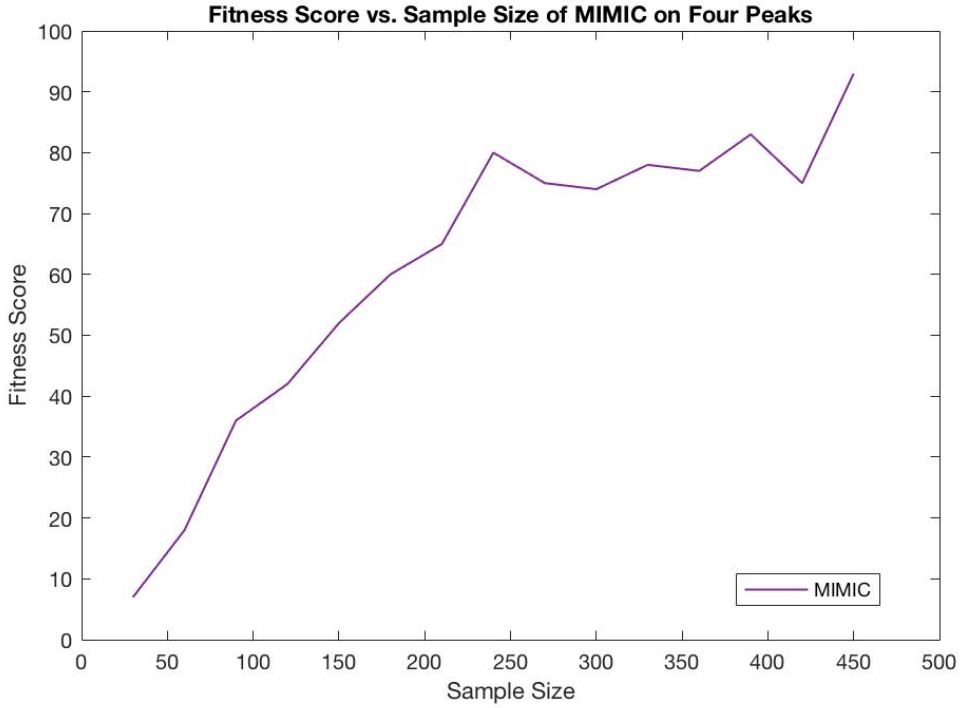

Figure 7: Fitness Score vs. MIMIC sample size on the Four Peaks problem

Looking at figure 6 , we see that MIMIC performs the best with the least number of iterations even though all algorithms manage to capture the maximum fitness score of 100 given enough number of iterations. The reason MIMIC is doing extremely well on this problem can be explained with its use of uniformly distributed random variables and scanning the entire area only to narrow it down for desired peaks. However, RHC and SA are greedy algorithms that get stuck in local peaks, and they needed to be restarted many number of times so they can end up in the right peak. However, it needs to be noted that MIMIC takes almost 10 times longer than GA.

Figure 7, shows results of the second experiment and it's show how the increasing sample size affects if the algorithm reaches to the right conclusion. As it can be seen from the graph, as the sample size increases fitness score increases as well, however, the rate of increase drops since after one certain point the possibility of having the right data points in the sample doesn't increase as fast.