# CS 4641 Project 2: Randomized Optimization

# Phase 1: Training a Neural Network

## Dataset for Neural Network training

From Project 1, I continue to use the political party voting dataset. As a recap from Project 1, the voting dataset classifies 1984 congresspersons as Republican or Democrats based on their votes. The data was stored in ARFF format, which ABIGAIL can import. The data contains 16 discrete binary features and 1 discrete binary label class (democrat/republican).

In order to actually create an optimization problem, a neural network was created with a cost function (Sum of Squares[[1]](#footnote-1)). This allows the following optimization algorithms to optimize the weights of the neural network (inputs to the problem) in order to minimize the cost.

## Optimization algorithm 1: Randomized Hill Climbing

Randomized Hill Climbing searches for maxima, which intuitively climbs hills by going to the maximal neighbor at each time step. In order to avoid getting stuck in local maxima, it randomly picks points in order to increase the probability that it reaches a global maximum. RHC[[2]](#footnote-2) is already implemented in ABIGAIL, which was used by the experiment runners. RHC is not guaranteed to find the global maximum and can perform poorly depending on the cost function.

## Optimization algorithm 2: Simulated Annealing

Simulated Annealing mathematically specifies the random behavior of searching for global maxima by modeling the process by which metals “anneal”. Temperature decreases over time reducing the probability that SA[[3]](#footnote-3) jumps to a sub-optimal point. SA can perform poorly depending on the choice of temperature decay and tuning of other parameters.

## Optimization algorithm 3: Genetic Algorithms

Genetic algorithms combine models in order to maximize some kind of objective fitness function. GA[[4]](#footnote-4) attempts to optimize the fitness function through crossover of models, which models natural selection. GA has several drawbacks in this optimization problem – the first is that optimizing a neural network is a continuous problem where weights are continuous variables. GA performs best on discrete optimization spaces. Moreover, sampling strategy and amount is very important, as it is important to capture a representative sample across the optimization space. If a combination of the samples in the initial space cannot produce the desired maximum, then GAs may perform sub-optimally.

## Results – Runtime

There is a very clear linear relationship of runtime versus iterations (). Per iteration runtime was lowest for SA and highest for GA. This aligns with what we may expect as GA is performing the most work (crossover and evaluation).

## Results – Average Performance

I ran 4 trials of the experiment with default values for parameters. The average score is plotted below (as randomized algorithms can have significant variance due to their randomized nature).

In the chart above, it is interesting to see how the different algorithms perform. RHC performs quite well and rapidly converges on an answer. Continued iterations fail to improve performance significantly. SA appears to vary considerably in the first 5000 iterations but appears to monotonically converge thereafter where it likely converged on a path near the global maxima. This result aligns with the functioning of the algorithm whereby it reduces the “temperature” as iterations increase.[[5]](#footnote-5) GA has a surprising result where it appears to not converge readily on the answer but instead only finds incremental performance improvement past the 5000th iteration. This behavior will be investigated further by varying parameters in a following experiment to study the effect of the population size on performance.

SA performed best in the end (at 50,000 iterations) but converged much more slowly than RHC. I suspect RHC was able to converge on an answer relatively quickly as compared to the other algorithms given the large size of the optimization space (16 features). This meant that randomly sampling throughout the optimization space was somewhat effective at converging on the global optimum quickly. Compare this to Simulated Annealing where it appears to struggle for a while to find a path to the global optimum but steadily converges on a solution after temperature decreased sufficiently.

One interesting aspect of the data is the structure of features. As there are a fair number of features where some provide a large amount of information gain (bills where politicians vote along party lines) as compared to relatively low value features. Given some “levers” that do little to influence the result (low information gain attributes), various algorithms have different responses. This is one reason that genetic algorithms may perform poorly as some of the “levers” in the model do not represent a separate dimension of the resulting data. This hurts the performance of GA as the initial population may represent the space of models poorly.

I suspect that there are a fair number of local optima, which would explain why GA and SA perform poorly. Factors that would contribute to this are the large number of features and the fact that some attributes roughly (but not exactly) correlate with each other.

## Results – Optimizing Temperature Cooling Rate

An experiment was conducted to determine the optimal cooling rate for Simulated Annealing. Through experimentation, I observed performance varied depending on the scale of cooling rate versus initial temperature (as compared to the raw values of each).

Iterations were fixed at 500 and 10000 in order to capture a snapshot-in-time picture of various values for the cooling rate. 15 trials were conducted and averaged to smooth out the resulting performance chart. Initial temperature is set to 10.

The results with this experiment are quite interesting – it appears that SA performs poorly with cooling rates very close to 1. This makes intuitive sense as this would make temperature decay slowly, which could cause SA to explore more perhaps sub-optimal areas. Interestingly, performance did not drop for lower cooling rates (closer to 0). These lower cooling rates should cause SA to converge faster. This implies that for cooling rate, there is a “good-enough” range of values for which it will perform. For values very close to 1, a large number of iterations are required in order to converge.

RHC seems to work well for the given optimization problem and a value of 0.95 for the cooling value seems to be optimal as it has more room to grow but still converges to an optimal solution relatively quickly. As the number of iterations, almost all values for the cooling rate except for those very close to 1 converged.

## Results – Optimizing Genetic Algorithms Initial Population Size

I expected larger populations to have better performance for GA compared to smaller populations as a larger sample should capture a more representative snapshot of the population. However, it appears that beyond some initial variation, performance is more-or-less constant as initial population size increases. This makes sense as the network is limited in terms of size of distinct parameters so adding more members to the initial population does not contribute significant new information or diversity.

It is interesting to see that there is overfitting with GA as an increased number of iterations results in improved train accuracy yet decreased test performance. It appears choosing a smaller initial population size is a method to combat parameters overfitting. From an optimization perspective, this is not as relevant to this paper as all the GA is trying to do is to minimize the cost function.

# Phase 2: Optimization Problems

## Problems chosen for optimization

### Travelling Salesman Problem

TSP is the problem of finding the optimal Hamiltonian cycle in a graph. This problem is NP-complete. Modern optimization techniques can approximate a solution within 2-3% of the actual solution. As TSP is implemented in ABIGAIL, optimization is performed by computing the sum distance of a Hamiltonian cycle and then optimizing over the inverse. I suspect GA will perform well on this problem given the structures of the underlying problem where I can see how combining two short paths has a chance at producing a new instance that may have a shorter overall path.

It is apparent when scores are normalized that GA is very well suited to TSP. This follows from the reasoning presented above where that TSP is easily digested into sub-problems and then combined (e.g. TSP solution for some partition of cities). GA combines smaller sub-paths through cities through various iterations in order to converge on an optimal path. I am surprised by the relatively poor performance of MIMIC compared to other algorithms. I suspect this is because the structure behind TSP is not easily modeled as a set of dependency trees. I also suspect that MIMIC and SA performed poorly given the large size of the optimization space ( paths for cities).

### Count Ones

Count Ones is a very simple problem that attempts to count the number of 1’s in a size bitstring. I suspect that GA and MIMIC will not perform as well as SA on this problem as it does not have much structure. I suspect SA will perform well as there are many local minima (as one can flip single bits in the middle of a bitstring).

MIMIC performed well alongside RHC but SA was very close behind. Interestingly, GA performs poorly at this task. With no structure, it is not surprising that GA performs poorly while RHC and SA both perform well. Dependency structures aren’t very helpful here, as mutating a bit shouldn’t vary the result too greatly. However, the runtime for SA was orders of magnitude faster than MIMIC. This highlights strength of SA or RHC – when problems lack structure, SA and RHC can efficiently reach a near optimal solution rapidly. Note that here, SA had less runtime than RHC so given how close resulting performance is, it makes sense to use SA over more iterations.

### Knapsack Problem

The Knapsack problem is where one wants to choose items that sum to a certain max weight limit while maximizing the value of those items. I suspect MIMIC will perform well on this problem as the Knapsack has structure in that the choice to add an item to the knapsack is conditionally dependent on the items in the bag currently. This structure can be approximated using dependency trees.

Not surprisingly, MIMIC performs very well on knapsack. There is clear structure as described earlier that is captured well by dependency trees. It is interesting that GA did not perform anywhere close to the performance of MIMIC. This is perhaps because more structure is needed than the mutation/natural selection structure of GA problems. RHC and SA are very close in performance. This makes sense, as they both do not exploit structure, which appears to be key in order to solve the problem.

1. With just 1 output class, the cost function was simply [↑](#footnote-ref-1)
2. Randomized Hill Climbing [↑](#footnote-ref-2)
3. Simulated Annealing [↑](#footnote-ref-3)
4. Genetic Algorithms [↑](#footnote-ref-4)
5. Given an increased number of iterations, SA actually slightly exceeds the performance of RHC. This is not included in the chart as it disturbs the axis scale [↑](#footnote-ref-5)