CS 7641 Assignment 2
Jeremy Martinez – jmartinez91@gatech.edu

Introduction to the data

The dataset used in this assignment is a list of restaurant reviews. The attributes are dictated by the description/comment with a binary classification of whether the review was positive or negative. The dataset contains 1,000 samples and is split evenly with 500 positive, 500 negative.

I faced new challenges running a dataset like this than when running it in assignment 1. The number of attributes significantly slowed down the training of the neural network. Computation was taking hours and so I was forced to scale back the number of attributes to just the 100 most common words. This did not have a significant impact on the accuracy of my randomized search algorithms, which I will elaborate on.

Neural Network

The neural network was trained using three different randomize search algorithms and backpropagation as a basis for comparison to what was used in assignment 1. The three search algorithms used include Random Hill Climbing (RHC), Simulated Annealing (SA), and Genetic Algorithms (GA). The input layer was set to 100. Only one hidden layer was used with a dimension of 40 and an output layer of 1. Each algorithm was run with 4,000 iterations at which point the algorithm approached convergence. Metrics for each algorithm can be seen in Table 1. Assume 4,000 iterations as a parameter for RHC, SA, and GA. Parameters for GA are in the order of [population, number to mate each iteration, number to mutate each iteration]. Parameters for SA are in the order of [starting temperature, cooling rate].

Algorithm	Parameters	Train Accuracy	Test Accuracy	Time (s)
Back Propogation		91.75%	67.5	155
RHC		71.25%	65%	64
SA	[1e10, .1]	75.75%	65%	36
SA	[1e10, .3]	77.5%	71%	37
SA	[1e10, .5]	80.5%	69%	64
SA	[1e10, .7]	74.5%	66%	65
SA	[1e10, .9]	74.5%	67.5%	61
SA	[1e6, .1]	78%	71.5%	43
SA	[1e6, .3]	76.25	66.5	42
SA	[1e6, .5]	77.3%	69.5%	42
SA	[1e6, .7]	75.875%	72.5%	42
SA	[1e6, .9]	77.5%	70.5%	38
GA	[45, 10, 10]	57%	56.5%	702
GA	[45, 10, 20]	57.125%	55%	868
GA	[45, 20, 10]	57.875%	57%	929
GA	[45, 20, 20]	60%	55.5%	1069
GA	[55, 10, 10]	53.625%	44.5%	612
GA	[55, 10, 20]	59.375%	61.5%	781
GA	[55, 20, 10]	57.5%	54.5%	846
GA	[55, 20, 20]	60.25%	56%	1025

Training times across GA took significantly longer than SA, RHC, and Back Propagation for little to no gain. When comparing the performance of each of these algorithms it is important to consider the test accuracy overall, as well as in the context of it's training accuracy. The contextual analysis demonstrates an algorithms ability to accurately generalize within it's own implementation (pertraining to overfitting). If an algorithm is much less susceptible to overfitting, then further optimizing this algorithm may be more valuable than tweaking an algorithm that is more likely to overfit.

Back Propagation (control)

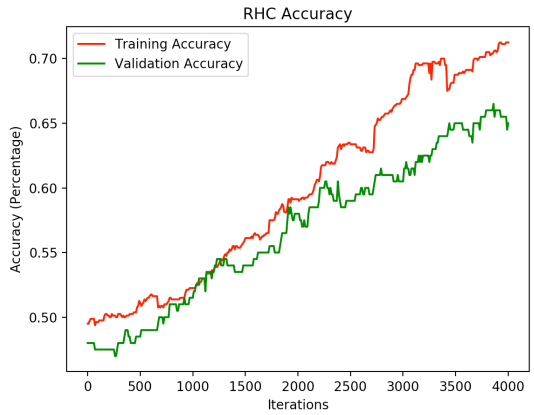
We use back propagation as a control variable to compare against our solution in Assignment 1. Since our analysis from Assignment 1 was on back propagation we won't delve too deep into this. It is worth noting the discrepancy in training accuracy vs test accuracy. This suggests (overwhelmingly so) overfitting. Despite the rather high accuracy percentage of 91.75% on our training data, back propagation does not generalize well. 67.5% is low in comparison to 91.75% and it is not even the highest test accuracy we found. However, it does still outperform RHC.

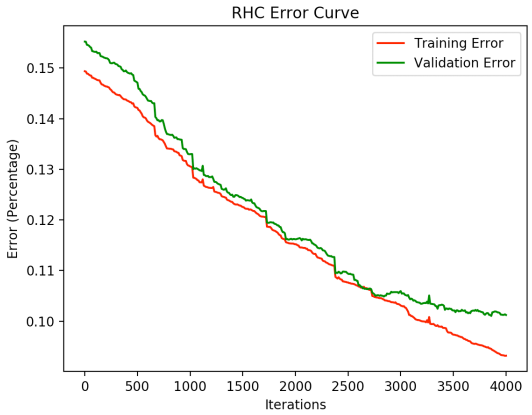
Random Hill Climbing

Random Hill Climbing is an algorithm that uses a concept of selecting a node at random, then checking a number of neighbors, selecting the highest out of all neighbors, and then repeating the steps until it has found a maximum. This algorithm is easily tricked into local optima since it does not consider anything outside the value of its neighbors. To avoid this, the algorithm is restarted several times at random locations. The idea of this is to cover the space.

This algorithm is an issue for my particular dataset since all of the features are binary. Comparing neighbors on a single feature is comparing 1's and 0's. This algorithm is more applicable to quantitative features that compare well. Binary comparisons are either true or false and so there isn't much of a "hill" to climb, so to speak. When comparing binary features in a hill climbing sense, we don't have continuous data that has the potential to (comparatively speaking) spike in value causing dramatic hills in our data. So binary data is relatively normalized across all our samples. When we visualize this (though hard to do so in 100 dimensions) we can think of the data never having dramatic optima, but instead several "valleys" and "peaks." This will cause an algorithm like RHC, one that quantitatively evaluates features, as more of an arbitrary comparison in respect to a binary feature set. This is an important quality to understand moving forward with SA and GA.

RHC trained an accuracy of 71.25% and had a validation accuracy of 65%. Both of these values are lower than the back propagation. We mostly care about the validation accuracy as this is the best indicator of the ability to generalize, though BP still outperformed it. We can analyze this validation accuracy in the context of our train accuracy and then it seems to be less susceptible to overfitting than BP. This metric can be useful to consider, and we will revisit this in our overall analysis. Accuracy and error curves for RHC can be found in Figures 1 and 2, respectively.





Simulated Annealing

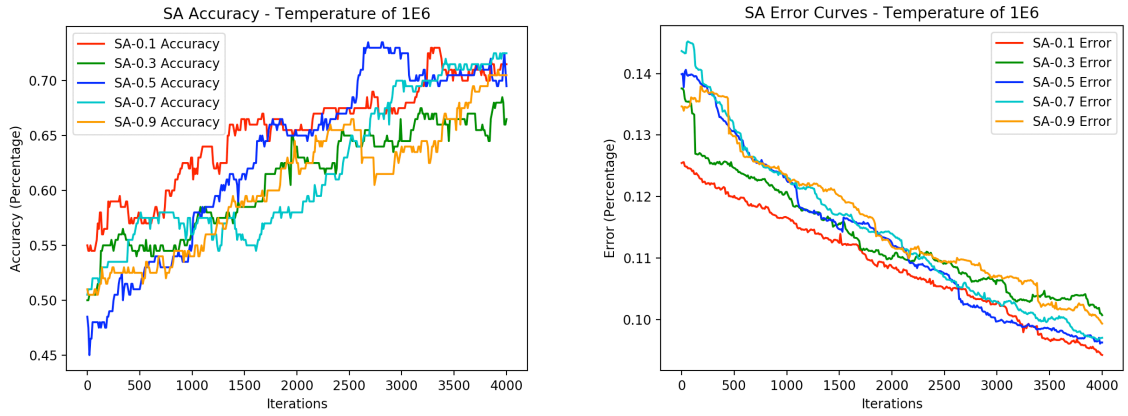
Simulated annealing is an algorithm similar to RHC in the sense that is explores its neighbors for optima. However, unlike with RHC that selects the highest neighbor no matter what, SA has a probability of selecting a neighbor that is not necessarily the highest. In regard to terminology here, we refer to

selecting the highest neighbor as exploiting. Since it is exploiting the neighbor with the highest value in these comparisons, the naming convention makes sense. On the contrary, the idea of choosing a lesser neighbor in hopes of finding a better optimum is a term we refer to as exploring.

The probability of whether the algorithm will exploit or explore is based on an initial rate we pass the algorithm, referred to as the temperature. This probability over the life of the iteration decreases. As the temperature decreases (referred to as the cooling rate), the algorithm approaches the strategy of plain RHC. The motivation behind exploring rather than exploiting is the belief that always exploiting may lead to the algorithm believing it's data too much (overfitting).

Exploiting resembling overfitting is interesting in the context of this specific dataset. Remember the analysis of our data we covered in RHC, "binary data is relatively normalized across all our samples." Our data will not contain dramatic optima, so exploring a new neighbor (rather than exploiting) could potentially place us in far away from any optima we were potentially approaching.

For SA we try the initial temperatures of 1E10 and 1E6. For each of these starting temperatures, we tested cooling rates of 0.1, 0.3, 0.5, 0.7, and 0.9. It's tough to nail down with absolute certainty what the optimal tuning of these parameters would be in a general case. Our best tuned parameters were [1E6, 0.1] while our second best could arguably be [1E6, 0.5], [1E6, 0.7], or even [1E10, 0.3] depending on how literally you want to scrutinize these metrics. Here we are defining best as the overall highest validation accuracy, meaning the algorithms ability to generalize well in combination with lowest error. Comparison of these validation accuracies can be seen in Figure 3.



The tuning of these operators in this case is something that should not be taken as a general rule of thumb. Just because [1E6, 0.5] did best here (generalized the best) does not necessarily mean we could use these parameters as our optimal tuned parameters for another set. The question of which parameter set is best implies the answer "it depends." Though I do believe our initial cooling rate of 1E6 collectively generalized better than 1E10. In deciding [1E6, 0.1] parameter set was the best was due to the fact that it's validation set peaked (at the highest) of 73% near 3,330 iterations (though it ended 71.5%). It also had the lowest error on the validation set of 9.4% (see figure 4).

Our SA algorithm did the best job of not overfitting as well. This observation is almost more valuable than anything in my opinion. On average, in respect to the training score, SA always did best in fitting the data accurately in respect to the validation set. This means that if we could continue to tweak these parameters we could (possibly) expect that as our training accuracy continues to climb we will see that

reflected in our validation (generalized) accuracy. It is also noteworthy to mention that out of all four search algorithms, SA had the lowest error curve.

Now that we have decided SA with parameters [1E6, 0.1] is most optimal for this specific dataset, we are free to explore take this (and maybe the next few closest parameter sets) and explore different preprocessed sets of the data. We can add layers, consider different feature selection/feature transformation, consider more words (more features). Since these algorithms take so long to compute, we want to explore these search algorithm and parameter sets first. Once we select a subset of what is most optimal to explore, we can continue to try to get our training/validation error down and accuracy up.

Genetic Algorithm

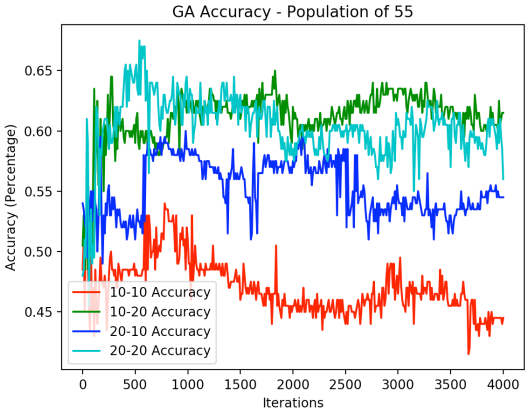
Genetic algorithms take a different approach than RHC and SA. In genetic algorithms, we throw away the notion of looking at neighbors and gradually moving in one way or the other toward finding optima. Instead, we take sets of features and tests their performance. Then we take the most N performant of these selections and mutate them on an M number of bits to (hopefully) generate more performant "offspring" of our original hypotheses (as well as cross over).

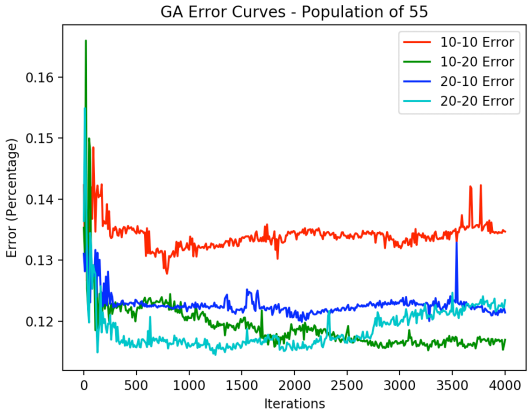
We can think of think of our mutations as neighbors of our original selection. Let's say, for example, our initial selection is 01101001 where each bit represents a specific word; 0 means this word is not found in the review and 1 means the opposite. In order to take mutations, we will find bit strings that are most performant and mutate an M number of bits to find possibly even more performant bit strings. We will repeat this process over a number of generations (iterations) much like the other algorithms.

GA also introduces a notion of cross over, which will allow us to take the combination of two selections (bit strings) and take a combination of the two. This will allow us to jump across the data set and find more optimal combinations of hypotheses. While one could view the mutation similar to RHC (especially when we only mutate on bit), however, cross over is more sporadic and allows for a more unique traversal of the dataset.

GA was extremely slow in this particular dataset. It was necessary to thread this algorithm just so that the permutations of parameters could all be explored in a timely fashion. For GA we used parameters ranging from a population of [45, 55], a mating rate of [10, 20], and a mutation rate of [10, 20]. These parameters were fairly close to one another and considering the long computation time, it did not seem informationally significant to explore further parameter tuning.

When considering these parameters, a population of 55 had higher accuracy and lower error across the board for the validation set, indicating better generalization. The accuracy curve as well as the error curve for the population of 55 across the rest of the parameters can be seen in Figures 5 and 6, respectively. When analyzing all of these parameters, having a mating rate of 10 and mutation rate of 20 appears to be the best. This conclusion is drawn from the fact that it has the lowest validation error and highest validation accuracy: best generalization. In the learning curve of the parameters [55, 10, 20] we find that the training converges to (nearly) the same error as the test 12.1% and 12.3%, respectively. This indicates that the GA algorithm is not overfitting and our fitting matches the validation set well.

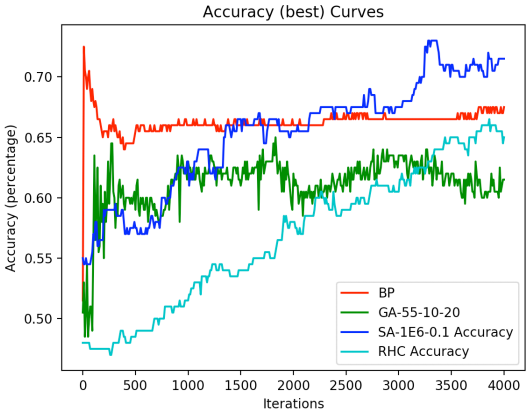


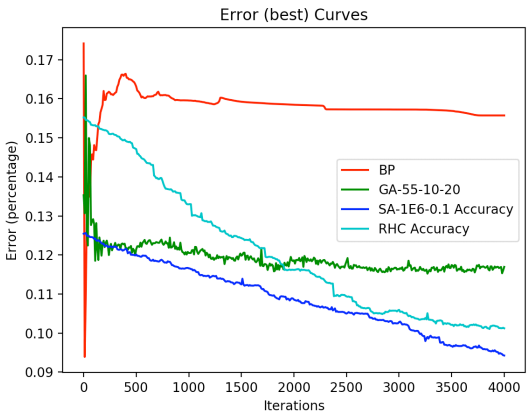


Overall, the genetic algorithm was computationally expensive and not performant. So while it is interesting to investigate, our data suggests with high certainty, that we would not want to use this search algorithm in further training of a neural network on this dataset. This could be due to the cross over introduced in this algorithm. Since our feature set is not quantitative at all, this could mean that GA does not generalize well over qualitative data. This would be an interesting characteristic to explore for future datasets

Neural Network Recap

When looking back over our four search algorithms, we can conclude that GA was by far the most computationally expensive, with BP second most expensive, then RHC and SA depending on the tuning of the parameters of SA. In Figures 7 and 8 we can see how each algorithm performed in accuracy and error, respectively. For each of these lines, we took the "best" tuned parameters that we concluded in each algorithms section above.





We ran these until 4,000 iterations. This is because in general we saw convergence by this number and GA was computationally slow even at 4,000. It would be worthwhile to see how the rest of the algorithms converge up to even 6,000 iterations. Maybe we'll see some increase in accuracy, though I believe this number of iterations correctly distinguishes these four algorithms sufficiently.

From these graphs, we conclude that SA was most performant in terms of overall generalization and resistance to overfitting. It had the closest convergence between its training and validation data. The GA accuracy curve seemed to have a lot of noise and converged lower than any of the other algorithms. RHC was still improving in accuracy at 4,000 iterations and so exploring this algorithm further in particular would be interesting to see if it eventually surpasses SA.

With this specific dataset, we are concerned about the curse of dimensionality. So we could see the most optimal gains in our feature selection and transformation. We could certainly do some interesting preprocessing of our data to weed out words that say do not have much of an effect on our algorithm. We could further analyze the relevant vs usefulness of our features use these analytics to enhance our feature selection. Next steps in further exploring these search algorithms in regard to this dataset would be to take SA and explore other transformations of our dataset that we are feeding the algorithm

Optimization Problems Overview

In our optimization problems, we analyze the flip flop, traveling salesman, and continuous peaks algorithms. We apply all four of our searching techniques to these and measure the fitness performance. Fitness is a measure to evaluate the optimum solution of the desired problem at hand. This is the reverse measure of a cost function and allows you to come to the same conclusion. What type of fitness function to use can have a big impact on the performance of your algorithm since it evaluates how any given solution is to the optimum solution.

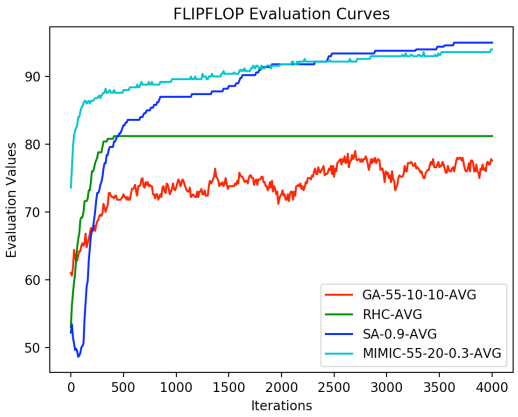
Here we compare our three optimization problems in respect to MIMIC, SA, GA, and RHC while tuning all the same parameters as before. In order to save space, in the tables below I will only enter the "best" row of each tuning of parameters per algorithm. Assume 4,000 iterations for all parameters. For order of parameters respective of each algorithm, refer to Table 1. MIMIC's parameters are in the follow order: [population, number of samples to keep, the smoothing constant]. Each algorithm with each set of parameters was run five times. Then the mean of those five outputs was used to calculate the result. For all of these algorithms, we define best to be the algorithm with the highest fitness evaluation.

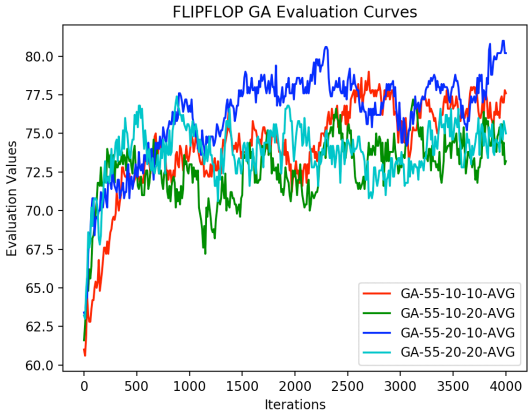
Flip Flop

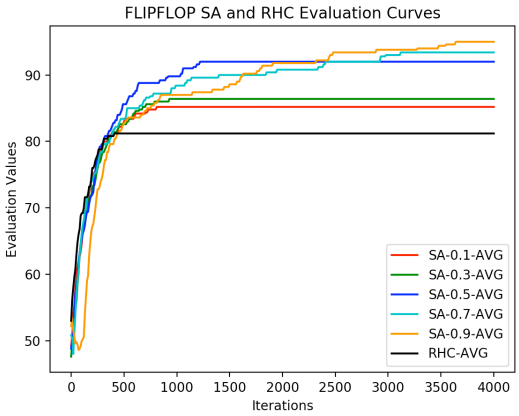
Flip flop is a problem that attempts to find the binary bit string (weights) that have the most alternating bits. In a bit string of length 8, the string 01010101 and 10101010 would yield the highest fitness evaluation. We see from Table 2 that SA and MIMIC performed the best when using the flip flop optimization as the evaluation function. However, MIMIC was significantly slower than all the other algorithms. When we factor this in, SA is the best algorithm when paired with this evaluation function.

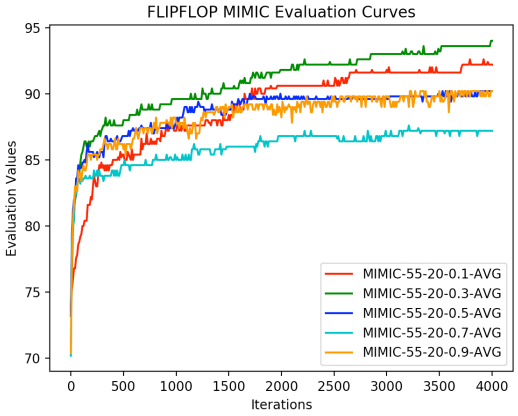
This evaluation function will tend to create several local optima making RHC and SA difficult to find neighbors with significantly higher values. These algorithms likely will get caught in local optima often. This is why the SA and RHC evaluation curves converge relatively early and at a lower rate (when SA has a low cooling rate). In SA, the rapid cooling temperature helps with this because it forces the algorithm to exploit far more often than explore in later iterations. The GA algorithm introduces cross over which will take these bit strings and flip them. This may explain the noise in the data and why it staggers up and down. Mutations and cross over behavior in this algorithm will cause sporadic changes in how flip flop evaluates the altered string, since the evaluation function will change on the bit strings it's cuts on. Consider the scenario where an 8-bit string is cut [1,1010,001]. Since it is only cutting between bits that are the same, this will result in the exact same evaluation of the offspring. An opposite example could show a scenario where it only cuts on the alternating bits, resulting is a drastic change in offspring. This could be why GA scores low and does not increase much after 1,500 iterations. Figures pertaining to the FLIP FLOP optimization and our four search algorithms can be seen in Figures 9-12.

Algorithm	Parameters	Evaluation Value	Time (s)
RHC		82	8
SA	1E6, 0.9	95	12
GA	55, 10, 10	81	1.2
MIMIC	55, 20, 0.3	94	103









While most of the algorithms converged by 4,000 iterations, it looks like GA and MIMIC may still be increasing. It will be interesting to explore this function until 6,000 iterations. It would also be worthwhile exploring alterations of parameters for SA and MIMIC similar to their highest scoring parameter sets

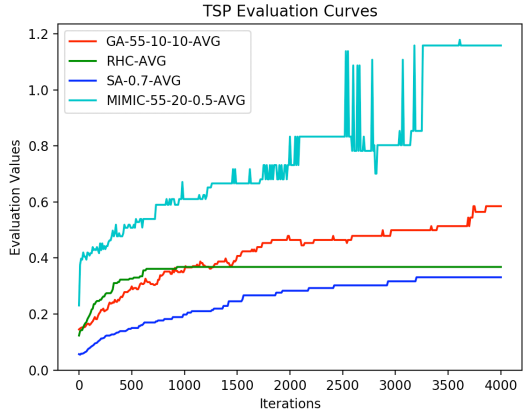
Traveling Salesman Problem

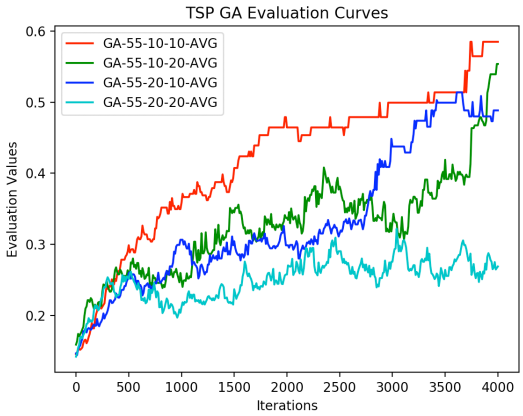
The traveling salesman problem is a problem that proposes the question: given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city? It is shortest possible path essentially. When I try to visualize this, I think of the single linkage clustering example from the lectures with k = 1, connecting similar strings based on a “closeness” calculation.

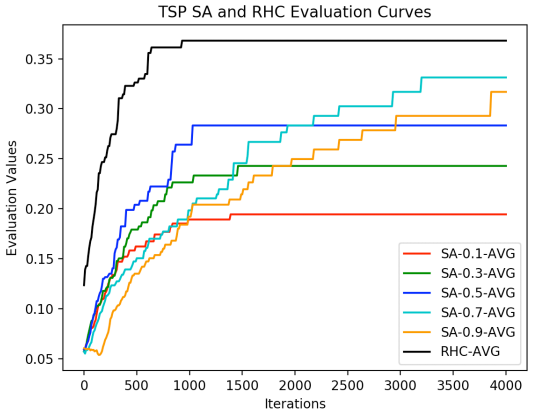
Initially I expected GA to be among one of the best algorithms for this type of problem. It takes mutations of each step and attempts to apply those mutations and crossover incrementally to find the shortest path. Then it didn’t do as well, though having a mating and mutation rate of 10 (the lowest) was most performant. This makes sense because it takes the smallest steps. So I reran it with parameters [55, (3, 6, 9), (3, 6, 9)]. It did better with the lower mating and mutation rates but converged near 0.72.

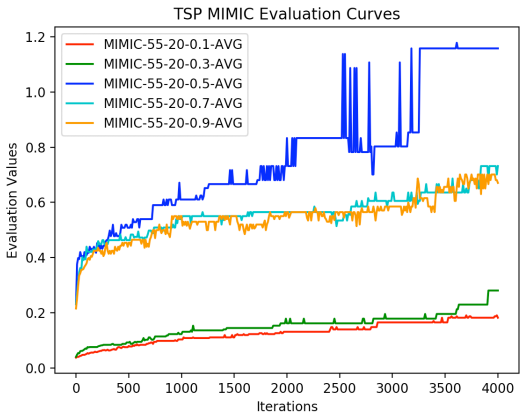
Then I looked into MIMIC and I believe MIMIC performs so much better here is because of it’s quality to be able to keep past iterations and avoid repeating bit strings it has already tried. This, combined with the smoothing constant allows for this algorithm to incrementally step through shortest paths without repeating itself and only consider values above what the smoothing constant establishes. Though, this requires several iterations and memory, which could be why we see a significant amount of time required to complete.

Algorithm	Parameters	Evaluation Value	Time (s)
RHC		0.367	0.05
SA	1E6, 0.7	0.33	0.024
GA	55, 10, 10	0.58	60
MIMIC	55, 20, 0.5	1.16	151









SA and RHC both converge early and low. I think it is safe to write them off on this optimization problem. I did run GA with lower parameters and saw some increase in fitness evaluations, however, they were not nearly as good as MIMIC. I also ran them till 6,000 iterations and they did not increase much after that either, which leads to me to believe they have (almost) fully converged here. The rest of MIMIC performs on par with GA with the exception of [55, 20, 0.5]. I am suspicious of these results and would want to dig deeper into if this was not just a computational fluke or not. Sometimes it is best to not trust your data!

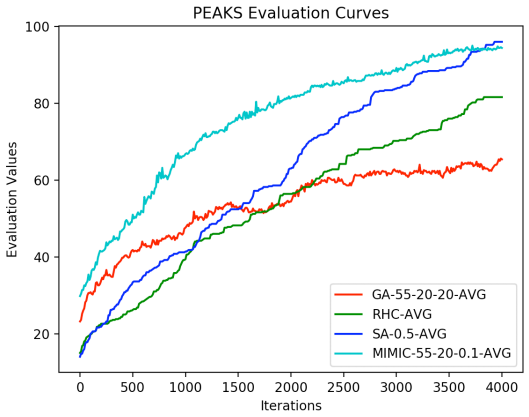
Continuous Peaks

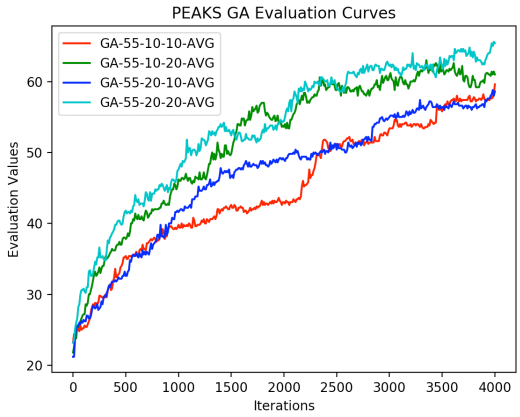
The continuous peaks problem is essentially a continuous version of four peaks. Four peaks searches bit strings that start/end with T-1 1's or 0's. Continuous peaks modifies this slightly to allow simply for T-1 contiguous 1's or 0's. SA and MIMIC both perform well with continuous peaks. Continuous peaks has optima of T-1 contiguous bits, which provides a high likelihood of optima. We set T to 49 in this case. This will not score strings with several local optima very high so SA and RHC naturally perform well here.

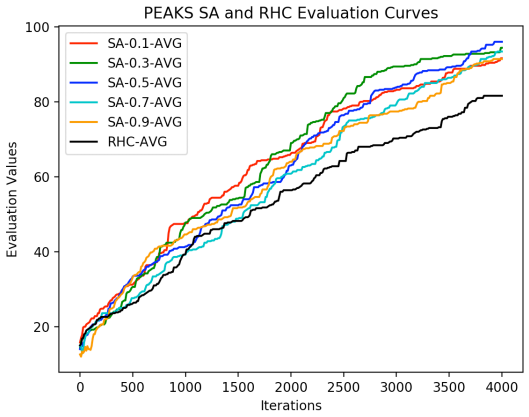
MIMIC performed very well with low smoothing constants and seem to converge in the low 90's. We use a population of 55 for MIMIC which is just above our T value. Having this values close to one another allows for the number of bits MIMIC saves to encapsulate T along with some other bits for strong

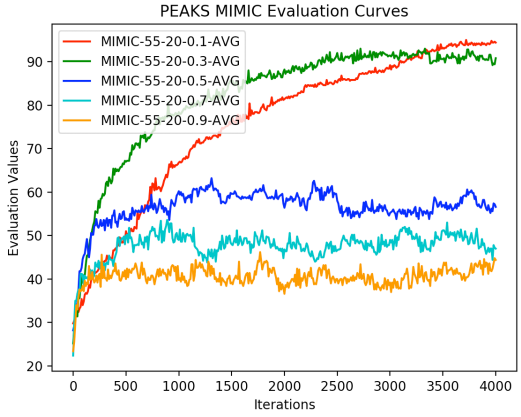
subsequent iterations. Since population is so high in respect to T, a low smoothing constant is why this performs best.

Algorithm	Parameters	Evaluation Value	Time (s)
RHC		81.5	0.02
SA	1E6, 0.5	95.75	0.036
GA	55, 20, 20	65.5	1.5
MIMIC	55, 20, 0.1	94.6	38.7









As we look at the curves for SA and RHC, it does not look like they are converging but rather continuing to climb. It would be interesting to continue these past 4,000 iterations to see where they converge

Overall Analysis

Testing these different optimization algorithms shows that it is worthwhile to test optimization that look for correlations as well as methods of representing and manipulating weight vectors that one may not anticipate working well with their dataset. We can use our knowledge of Flip Flop, TSP, and Continuous Peaks to rationalize success/failure in their application to specific searching algorithms. And this knowledge can in turn guide our hand in optimizing where we begin to build a Neural Network in the future. This information is extremely useful and applicable in re-world application. However, I think it would also be foolish not to explore these other (what may seem extraneous) optimizations in hopes that we find one that works when paired with a search algorithm

Writing code in a DRY modular way such as I have for this assignment, it is easy to plug and play with different search algorithms/fitness evaluation functions. This is a practice I will continue moving forward in applying learners to data and NN in the future.

Further Analysis and Wrapping Up Results

We see our best optimization problem/search algorithm pairs are FLIPFLOP/SA(1E6, 0.9), TSP/MIMIC(55, 20, 0.5), and CONTINUOUS PEAKS/SA(1E6, 0.5). When analyzing success of SA in comparison to RHC, it sheds light on exactly where SA is gaining improvement. If SA has a temperature of 0 then it behaves exactly like RHC. We find that with too high of a temperature, the algorithm to act in a random sporadic manner. As temperature starts lower and cools quickly, we see much more exploring up front and a quick shift towards RHC-like behavior. This is behavior proves to be most optimal with FLIPFLOP. For PEAKS, we see a similar behavior to this, though with a slower cooling right (and so more exploring while searching for optima).

Since PEAKS scores bit strings with T contiguous bits, the bit string we search over has more significant optima. This is why SA can perform better on PEAKS with a slightly lower cooling rate (more exploring). Despite it exploring more, since the optima curves are more severe, we can trust the searching algorithm finds the optima easier and so the optimization problem benefits more from exploring without taking a loss in fitness.

Looking over GA's performance, TSP seems to be the only one it performs well on. It makes sense why GA doesn't thrive in PEAKS because PEAKS represents the bit string in a way that is more compatible with a hill climbing like search. TSP looks for shortest path between bit strings. Mating and mutating bit strings similar to each other is a natural way of finding bit strings closest to each other. In our analysis of GA on TSP we discuss this. I even take this notion and tune the parameters more towards lower mating/mutation rates in order to gradually find similar bit strings and we see a increase in performance. In exploring these parameters further and continuing to see improvement, it feels like we have tapped into why GA performs well on TSP. This is one parameter set I would like to explore further.

As for MIMIC, it performs well in all our examples, though at a cost. MIMIC requires high computation and memory complexity. This is a trade off we need to consider, and in cases it does similar or even slightly better than other search algorithms, it may be better to explore the other algorithm and save the computation time. MIMIC performs so well in each of these because it's ability to prevent redundant searches. It uses the probability distribution to and only considers fitness functions that pass a certain threshold. By tweaking this threshold (referred to as smoothing constant above) we are able to explore only fitness functions that pass this threshold. With a high threshold our distribution is only over the optima (that qualify according to our threshold). With a low threshold, we assign uniform distribution over the space. Since this has to sample fitness across the space each time and then analyze these and iterate, that is why MIMIC is so slow (yet powerful)!

Tying it All Together

The randomized search algorithms explores different method of finding optima in our data. This is tightly coupled to the structure of the data that we pass in. Our data in the neural network is represented as a weight vector (bit string). We have to test these solutions to come up with the best solution for each given problem. We score these with the fitness function. As we have seen, some fitness functions perform better when coupled with search algorithms. So we are selecting these as a pair when ultimately passing our data through the neural network. We see here, in our limited experimentation with parameter tuning, which fitness functions perform well with which search algorithms. Next we need to tie them all together and plug our data into a neural network using our selected randomize search algorithm combined with its respective fitness function