Nilay Mehta

CS 4641 – Machine Learning

3/11/2018

Randomized Optimization

Introduction and Recap: The purpose of this paper is to explore four randomized optimization algorithms: randomized hill climbing, simulated annealing, genetic algorithm, and MIMIC. This will be achieved by presenting our analysis into two sections. The first section of the paper works with the neural network we optimized in the previous analysis paper and further tries to optimize its weighs using the randomized optimization algorithms listed above. In the second section, we will discuss three optimization domain problem domains and use our findings to analyze the benefits and advantages of each of the four algorithms for different problem sets.

- **Human Resources Analytics dataset -** This dataset explores had a variety of characteristics targeted to explain why employees leave prematurely from their occupation. It proposes the classification problem of identifying potential employees that hare likely to leave next through a series of attributes, some of which are the employee's last evaluation, average monthly hours, history and time spent with the company, the department they work in, and salary range. There is a grand total of 14,999 data entries with 10 unique attributes each, but to ensure reasonable training times for my experiments, I sampled various dataset sizes incremented by 600 data points ranging from 600 data entries up 3000 data entries. I chose this dataset for is assignment out of my two because it has a comfortable size of attributes. I also plotted the distribution of the attributes and, because many of them have multiple classifications and the frequency and distribution of attributes also varies significantly.

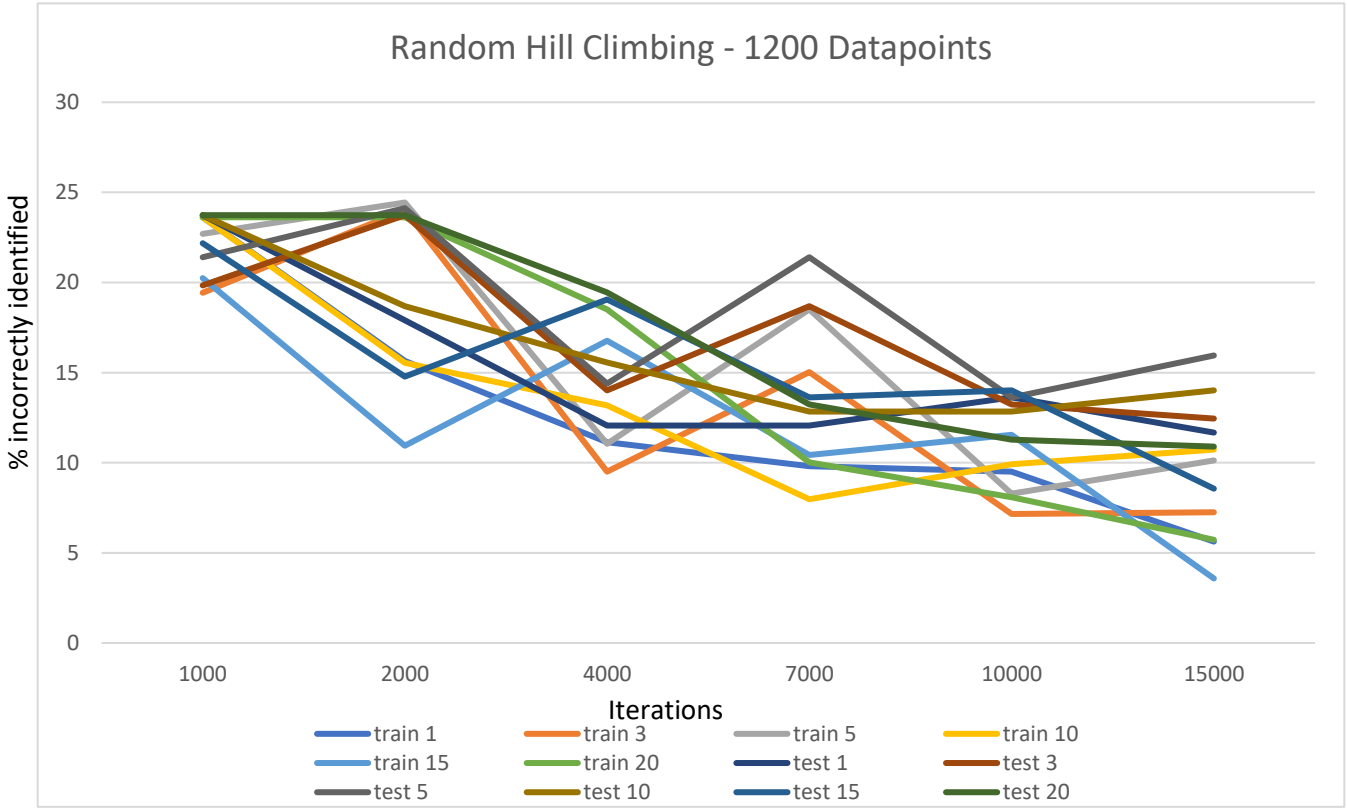
In my previous paper, we found that when preforming multiple experiments to attempt to optimize the neural network, we found that using three hidden layers with 22 to 10 neurons in each layer achieved the highest performance with a mean square error of 0.026 at training batch sizes of around 7000 for the HR Analytics dataset. Across the datasets, when tweaking the hidden layers there was a diminishing return with adding more layers and more neurons in each hidden layer, while the training time increased a significant amount. When adding four or five hidden layers opposed to the three hidden layers I finalized on, increased the accuracy a negligible amount. Neuron count had the same effect. The two configurations I tested for neuron count within the hidden layers were uniform count across all hidden layers, and a tapering count where the first hidden layer had the most neurons, and the last hidden layer had the least. I found that the uniform neuron layout worked better for the HR Analytics dataset. It is important to point out that my testing found that significant amounts of data greatly increased the performance of the algorithm. However, for the sake of time and my sanity, I limited my largest batch size to 3000 data entries, so our performance relative to the optimal neural network is limited.

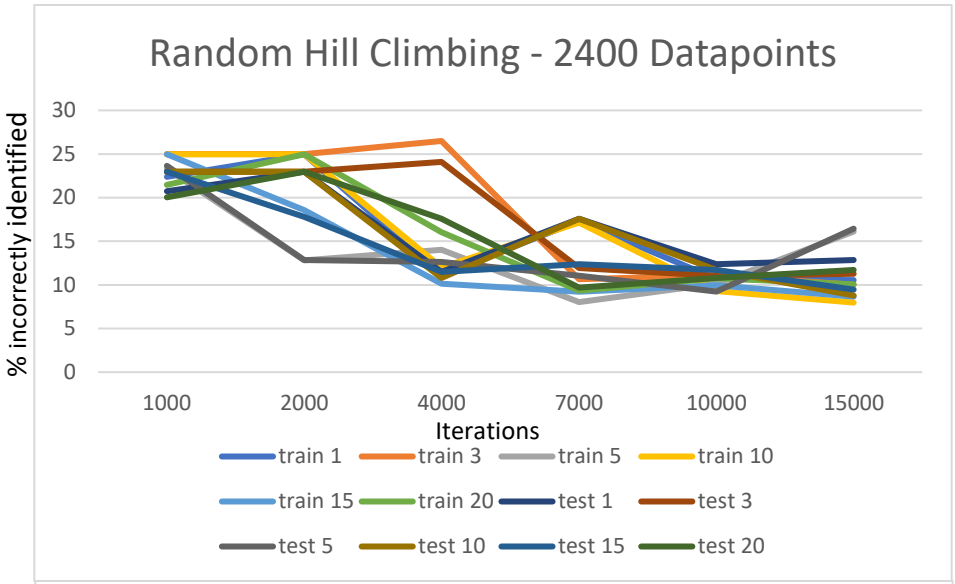
Further information, including the original linked to the dataset, can be found in the readme file. All the algorithms and analysis have been run while the data was partitioned following a 80/20 ratio for training and testing sets respectively. For runs that varied the training sample size, the testing sample set was still maintained at its original 20% size. Each algorithm was run multiple times with nuanced hyperparameters to determine the effects of each of these. The overall purpose of these tasks was to find the hyperparameters that optimized the accuracy while observing training characteristics like over/underfitting. The metric reported in the graphs to evaluate the performance in both the training and testing sets was percent of incorrectly identified entries.

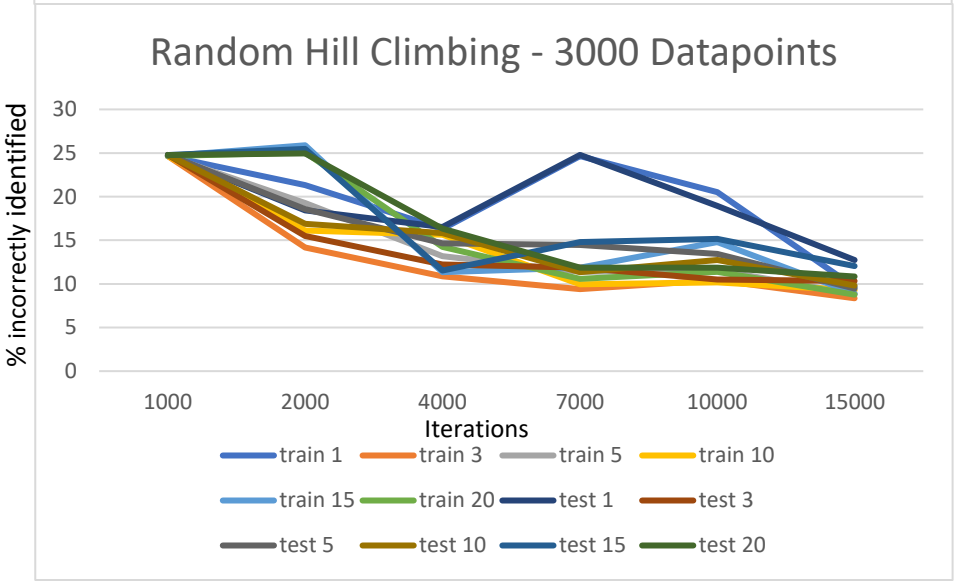
Part 1: Algorithm Runs on HR Analytics Dataset

Randomized Hill Climbing with Random Restarts

Random Hill Climbing is the simplest algorithm of the set we will be looking at in this paper and it works by picking random point, assessing the fitness of the neighboring points, going to the point with the best fitness, and then repeating the process. This algorithm repeats this process for the number of iterations until it finds a point with the best fitness. We need to take this 'optimal solution' with caution because, while it could be the optimal solution for the function, it is also possible that this pot is the highest fitness point, a point with the lowest cost, or we are stuck at a local optimum. The hyperparameters I observed were the number of restarts (ranging from: 1, 3, 5, 10, 15, 20) and the iterations the algorithm ran for. The algorithm also







ran over multiple dataset sizes, ranging from 600 to 3000 data entries, with increments of 600 each time. The goal of the random restarts was to limit the likelihood of a run getting stuck at a local optimum. The graphs seen on this page include both the train and test error for the number of random restarts preformed for their respective dataset size.

I was surprised to find that even across multiple dataset sizes, the number of restarts had a minimal effect on the overall performance. For each dataset size, there was a clear downward trend in the number of instances incorrectly identified and the hyperparameter that seemed to influence this decline the most was the number of iterations the algorithm ran. The number of restarts seemed to have negligible effects for each of the dataset size; the data suggests that three random restarts was sufficient to ensure that an optimal solution was found. A potential word of caution to this number might be influenced by the relatively small size of our dataset.

Another interesting characteristic of this algorithm was that it seemed to show noticeable improvements for both train and test error when jumping from the dataset with 1200 entries to 2400 entries, but the error increased for test error in the dataset with 3000 entries, a sign of overfitting. It is also significant to note that this was the fastest algorithm to run out of the set we will discuss in this paper. As a relatively inexpensive algorithm, the performance was not as impressive as our initial neural net (again, most likely due to the training set sizes), but decent performance was achievable with an expected error in classification of about 11%.

Simulated Annealing

Simulated annealing is a randomized optimization algorithm that builds off random hill climbing. It has characteristics influenced by the concept of annealing with metals, where the temperature and cooling is controlled to maximize the strength of the metal. Similarly, simulated annealing takes parameters like temperature and cooling rate to achieve something similar with our dataset. At its core, the algorithm starts at a random point and starts the first iteration by sampling another point within our dataset. The distance between these points is a probability distribution, and depending on the fitness function, it will determine if we want to move toward that point depending on if it moves us to our goal. One inherent advantage of this algorithm over random hill climbing is that because it relies on a distribution that have local optimas, the algorithm can decide to move away from the point to avoid becoming stuck in a local optimum. It also builds off of randomized hill climbing because with more iterations, if is able to explore the entire search space much more effectively rather than being limited to a certain section.

The data I collected, shown below, was for a sampled dataset size of 3000 entries that ran over both 2000 and 7000 iterations (only the data for 7000 iterations is shown). The first row of

Simulated Annealing with Dataset size = 3000 and iterations = 7000

Temperature (1E) \ Cooling rate(%)	50	55	60	65	70	75	80	85	90	95
10	11.53184	14.62995	13.76936	14.80207	13.76936	13.59725	14.1136	11.70396	13.25301	12.56454
	9.057072	12.15881	11.70389	11.20761	12.24152	10.8354	13.689	8.395368	11.37304	12.03474
11	14.80207	10.32702	16.69535	9.982788	12.04819	17.90017	12.56454	13.25301	11.53184	13.76936
	12.24152	9.346567	15.46733	9.346567	9.594706	14.92969	11.41439	10.71133	9.925558	11.16625
12	24.78485	10.84337	11.87608	12.39243	15.1463	11.70396	13.76936	13.0809	12.04819	12.39243
	24.68983	8.643507	9.511993	11.41439	12.11745	8.560794	12.11745	11.53846	8.684864	9.057072
13	11.53184	12.73666	24.78485	12.39243	13.59725	14.62995	11.70396	13.42513	14.28571	12.90878
	10.13234	10.09098	24.64847	9.594706	10.25641	12.03474	9.966915	11.53846	10.38048	11.62117
14	17.38382	10.84337	9.982788	14.80207	13.25301	10.84337	13.76936	16.69535	17.55594	8.605852
	16.41853	9.884202	10.42184	11.20761	11.57982	8.602151	11.49711	17.03888	16.45988	9.346567
15	12.39243	17.2117	12.22031	13.0809	15.49053	12.56454	12.39243	12.22031	15.49053	13.25301
	12.69644	16.12903	11.16625	11.37304	15.09512	10.50455	10.42184	9.842845	11.37304	10.1737

entries(highlighted) for each temperature is the training error and the second row is the test error. I varied the cooling rates starting from .50 to .95 over increments of .05 (lower values indicate quicker faster cooling across iterations) and ran those values over various starting temperatures (a decrease in the temperature limits how much the algorithm can jump between points). It's important to note that the lower the temperature, the closer the subsequent points will be and the likelihood of getting stick at a local optima increases.

It was difficult to come up with a graphical presentation of the data that was as concise as the table, but observing the values does give us a few insights on the performance of these hyperparameters. When there was a smaller number of iterations (2000), we observed a lower overall performance with an average train error of 18.69% and an average test error of 17.99% for the table. One observation I noticed was when there was a high temperature value, we would be able to reach a global maxima by jumping over the local maxima, but the low iteration count to cool down also means the randomness we gain from the high temperature value does not guarantee a local maxima. Lower cooling rates or a lower initial temperature can reach a local maxima with a smaller amount of iterations, but this issue is not present when we pushed the iteration count to 7000. For the higher iteration run, we observed fairly consistent performance across most of the temperatures. There were however local optimas at different cooling rates. As a general rule of thumb, higher temperatures seemed to have lower cooling rates that were optimal. A potential explanation is that the initial high temperature allows the algorithm to explore the dataset more thoroughly and the lower cooling rate causes the temperature to decrease at a higher rate, which in turn decrease the ability for the algorithm to jump across the dataset resulting in closer subsequent points. But, because the algorithm initially is able to jump more widely and assess if it is optimal, this combination compliments each other and is able to perform well. The overall performance of this algorithm was adequate. It did not perform as well as random hill climbing, but the average test error at 7000 iterations was relatively close at 12.11%.

Genetic Algorithm

Genetic Algorithms are inspired by the evolution of biological systems and works by modeling weights to our neural network as singular individuals in our population, we assess the performance of these individuals and take a certain amount to "mate" with each other over a set number of iterations to produce new individuals with the goal of taking the best traits of both individuals and inmoving it over successive generations. We can also model mutations, which are randomized changes to parts of the population with the goal of increasing genetic diversity and ensuring against a homogeneous population (which would result in getting stuck at a local optima).

The hyperparameters for this algorithm is the initial population of individuals, the number of new individuals in each generation denoted as mates, and the number of individuals that have mutations in their weights. The values that were tests were with populations of 100 and 200, 25 or 50 mates, and 0 or 25 mutations for each generation. The data was run with a dataset size of

Population	Mate	Mutations	Errors
100	25	0	23.94366
			24.26319
		25	23.94366
			24.26319
	50	0	23.94366
			24.26319
		25	23.94366
			24.26319
200	25	0	23.94366
			24.26319
		25	23.94366
			24.26319
	50	0	23.94366
			24.26319
		25	23.94366
			24.26319

1800 entries with 500 iterations and is presented on the next page. Consistent with previous tables, the first, highlighted entry is the train percent error while the subsequent value in each section is the test error.

Unfortunately, despite playing around with the values and trying to produce errors that were not the same across all my runs, I was not able to. The algorithm had the same performance over all the hyperparameters. Initially, I thought it was due to my high iteration count or another hyperparameter, and decreasing any of these values did change the error, but the same problem persisted: all the train and test errors were the same for their respective fields.

Another surprising result of this experiment was that the errors were also consistently high; even after tweaking the values, I was not able to get a consistent error under 18% for both the train and test errors. However, I still would discuss the relationships I would have expected if my experiments produced better data. For the hyperparameter of population, there are two potential outcomes: we can either converge to the optimal solution slowly (because the other hyperparameters would result in an overall limited gene pool), or the population would become homogeneous and get stuck in an optima. Large populations, while preferable, also run the risk of having too many individuals with too much variability between individuals. This would be dangerous because there it is hard to distinguish an optima and result in sometime more in line with a random search. Mates are an important hyperparameter because they ensure against homogenous populations. A value of 0 or potentially in our case something like 10 would result in the population becoming homogeneous quicker than the previous algorithms. The hyperparameter for mutations achieves the same effect; it prevents against a homogeneous population and ensures that a more optimal solution can be reached. The overall performance of this algorithm was the worst out of the three, but this is mainly due to the issues involved with tweaking the hyperparameters and achieving varied results. The overall runtime of these experiments was a significant step up from the previous two algorithms and even compiling this small results table took nearly as much time as all of my randomized hill climbing experiments.

Part 2: Optimization Problems

Traveling Salesman (Genetic Algorithm):

The traveling salesman problem is a graph search problem where the goal is to find a graph traversal that travels to all N nodes but returns to the starting node with the lowest distance traveled. I ran all four randomized optimization algorithms using the traveling salesman problem with 50 nodes across a variety of hyperparameters. The performance of the best hyperparameters is summarized in the table.

ALGORITHM	HYPERPARAMETERS	AVERAGE OPTIMAL FITNESS (10 TRIALS)	AVERAGE TRAINING TIME
RANDOMIZED HILL CLIMBING	Iterations = 15,000	0.12459346222434765	0.133565318 sec
SIMULATED ANNEALING	Iterations = 15,000, Temp = 1E12, Cooling Rate = 0.95	0.09844102622544884	0.120167991 sec
GENETIC ALGORITHM	Iterations = 1,000, Population = 200, Mate = 100, Mutations = 20	0.14839679803874492	1.640889937 sec
MIMIC	Iterations = 1,000, Samples = 200, K = 100	0.08860896658677894	14.889664049 sec

The best preforming algorithm is the one that is able to return to the starting node with the lowest distance, and out of the optimal values, the genetic algorithm seemed to perform the best when looking at the average optimal fitness results. This makes sense because genetic algorithms thrive when the entire set of potential situations is visible, and it is able to select and combine the best preforming solutions from generation to generation. The genetic algorithm did take more time relative to randomized hill combing and simulated annealing, but the performance is significant enough to justify the marginal time premium. Randomized hill climbing performed well, but that is due to its inherently greedy approach when exploring one section of the search space. MIMIC looks at the probability distributions that there is a true maximum in an area and because the search space for this function was fairly large with many potential local maximums, I think it would need more than the 1,000 iterations to produce a more optimal solution (which would require a training time much higher than any of the other algorithms).

Flip Flop (Simulated Annealing):

Flip Flop is a fitness function that works by taking a bit string and returns the number of times the bits alternate in the bitstring. A bitstring with maximum fitness would return the length of the original passed bitstring (a bitstring that had completely alternating values). After running all four algorithms, we were presented with the results presented on the previous page. The bitstring length I passed in was

ALGORITHM	HYPERPARAMETERS	AVERAGE RESULT (50 TRIALS)	TOTAL TRAINING TIME
RANDOMIZED HILL CLIMBING	Iterations = 15,000	144.93	0.269955 sec
SIMULATED ANNEALING	Iterations = 15,000, Temp = 1E10, Cooling Rate = 0.95	171.70	0.65276761 sec
GENETIC ALGORITHM	Iterations = 1,000, Population = 200, Mate = 100, Mutations = 20	148.17	6.5532087 sec
MIMIC	Iterations = 1,000, Samples = 200, K = 100	153.89	967.76346 sec

180, so the most optimal values to return would be close to the maximum of 180. All the algorithms preformed relatively similarly except simulated annealing which preformed significantly better than the other algorithms, both with respect to the average result and training time. This is logical given that the fitness function is relatively simple and fast. This poses issues for the genetic algorithm and MIMIC which are slower, while randomized hill climbing is fast but because of the high amount of local optimas, simulated annealing is ideal with its idea of controlled cooling and temperature. This is an ideal problem for simulated annealing because as it keeps jumping around in the search space when the temperature is initially high, it is able to test multiple configurations and avoid getting stuck in most local optima, as mentioned above for randomized hill climbing. For these reasons, simulated annealing is an ideal candidate for this type of problem and performed better than the other algorithms.

Four Peaks (MIMIC):

Four Peaks is a optimization problem that takes in an input of a bitstring of length N and a trigger point T where $0 < T < N$. It has a global maximum where it has all 1s and 0s until the trigger point T, then all the opposite bits leading away from that point. There are four local

ALGORITHM	HYPERPARAMETERS	% OF RESULTS OPTIMAL (OUT OF 50 TRIALS)	TOTAL TRAINING TIME
RANDOMIZED HILL CLIMBING	Iterations = 15,000	0%	0.40022205 sec
SIMULATED ANNEALING	Iterations = 15,000, Temp = 1E12, Cooling Rate = 0.95	14%	0.28424435sec
GENETIC ALGORITHM	Iterations = 1,000, Population = 200, Mate = 100, Mutations = 20	6%	4.04604265sec
MIMIC	Iterations = 1,000, Samples = 100, K = 5	72%	155.61085305 sec

maxima in this problem and the values I used for the problem are N = 100 and a trigger point T = 8. The data collected and the hyperparameters are listed in the table above. Randomized hill climbing performed the worst, which is not surprising considering the complexity of the problem, yet this algorithm only explores very little of the search space and will always settle to the local optima. All the trials returned suboptimal results which makes sense because it is a greedy algorithm that doesn't have any way to share information regarding its results between instances. I was surprised by simulated annealing's performance; despite being able to explore more of the search space than randomized hill climbing, it was able to return 7/50 instances with optimal results. This could potentially be because of the relaxed trigger value but it would be interesting to see the performance if T was increased. The genetic algorithm was disappointing because I expected that there would be a higher rate for optimal results; I thought it would be able to have optimal learners that mated multiple times to result in a better weighted solution. While researching this optimization problem, I came across Dr. Isbell's paper on MIMIC and its performance was consistent with what is presented in this paper. MIMIC was by far the most optimal algorithm for this type of problem with 36/50 results returning optimal results. MIMIC was effective when considering was able to reach an optimal solution with a lower iteration count, but it still remains very computationally expensive. It is able to create a distribution of likely locations of local optimas and repeating this process feeds into a higher likelihood of finding the local optima, another reason MIMIC performed exceptionally well in this algorithm. One important feature of the algorithm to note is that it does not work effectively with functions with many local maximums or minimums, like Flip Flop, because the probability distribution would be more uniform and detrimental to MIMIC's performance because it would be hard to improve on a pervious iteration's performance.

Conclusion

Overall, each randomized optimization algorithm did a proficient job for classifying both the HR Analytics dataset and the three optimization problems relatively accurately. In the first section of this paper, the randomized optimization algorithms were not able to outperform backpropagation we used in our neural network in our first project, but they came respectively close. One potential reason for this is that the labels for the dataset is fairly skewed and biased. I was also disappointed by my results when I ran my genetic algorithm on the HR Analytics dataset. Despite tweaking the iteration count, population, mating, and mutation hyperparameters, I was not able to get results that showed variability in results, which forced me to draw conclusions from our discussion in class and the traveling salesman optimization problem. While some of the results were a little disappointing, the procedures and insights in the first section proved to be useful when solving the three optimization problems. It helped cut down the hyperparameter values I needed to test to most closely approximate the most optimal solution. Each of the three optimization problems (traveling salesman, flip flop, and four peaks) demonstrated the benefits of using genetic algorithms, simulated annealing, and MIMIC respectively and provided additional insights. The second section experiments were personally by far more insightful to see the strengths and weaknesses of each algorithm in different problem spaces.