David Strube - dstrube3@gatech.edu

CS 7641 - Machine Learning

Assignment 2 - Randomized Optimization

2020-10-11

Part 1

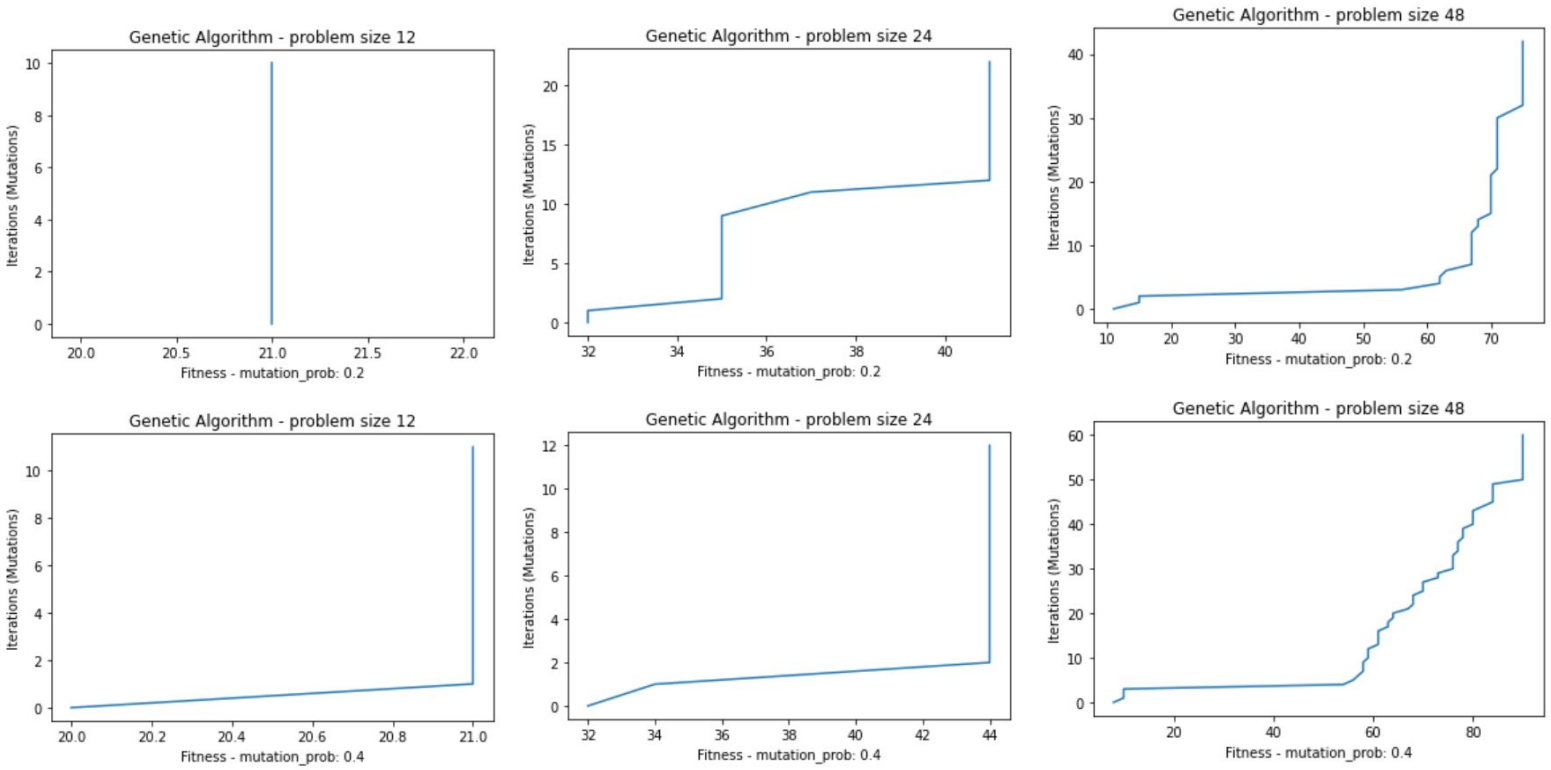
For the first part of this assignment, I implemented these four local random search algorithms: randomized hill climbing (RHC), simulated annealing (SA), genetic algorithm (GA), and MIMIC. I implemented them by using their implementations in mlrose_hiive (mlrose). In my implementations, I added code to explore a parameter in each algorithm. For RHC, the parameter was *restarts*, ranging from 10 to 40; for SA, it was *schedule*, covering all 3 types available (GeomDecay, ArithDecay, and ExpDecay, using all default parameters for each); for GA, it was *mutation_prob*, ranging from 0.2 to 0.4; and for MIMIC, it was *pop_size*, ranging from 100 to 300. In each implementation, I set *random_state*=1. (I was going to set *random_state*=0 like I did in Assignment 1 using sklearn, but in mlrose, random_state must be a positive integer.)

For my "created" optimization problems, I decided to use the following discrete-valued problems that come with mlrose: FourPeaks, Max K Color, and Knapsack. All four of the aforementioned algorithms were applied to each problem. The code and results of these algorithms being applied to these optimization problems can be found in "*A2 - Randomized Optimization - Part 1 - dstrube3.ipynb*".

Problem 1 - FourPeaks

For the FourPeaks problem, I gave each algorithm problem sizes of 12, 24, and 48. MIMIC performed well at all three problem sizes for all three of its population sizes. RHC did not do as well for all three problem sizes. At problem size 12, it converged on fitness 12; at problem size 24, it converged on fitness 4 (at *restarts*=10 and 20) and 6 (at *restarts*=40); at problem size 48, it converged on fitness 2. SA performed well at problem sizes 12 and 24 for all three schedule types. At problem size 12, it

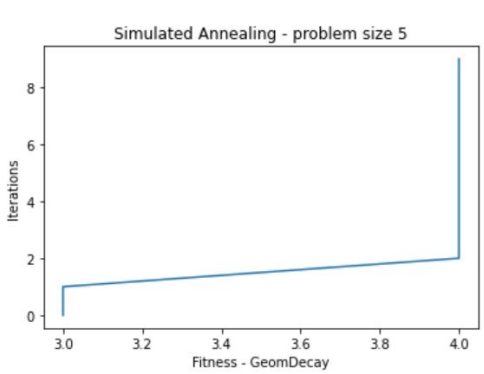
converged on fitness 7; at problem size 24, it converged on fitness greater than 30, which is not as great a fitness as what MIMIC found, but it did so in much faster time. However GA did best, finding fitness of 21 at problem size 12, fitness of about 44 at problem size 24, and at problem size 48 it converged on fitness of more than 70 at *mutation_prob*=0.2 and fitness of more than 80 at *mutation_prob*=0.4. The FourPeaks problem highlighted the advantages of GA because this problem's solution involves finding a reward based on a parameterized value T (defined by o(x) (number of contiguous Ones) and z(x) (the number of contiguous Zeros)) as well as maximizing o(x) and z(x). (Baluja and Caruana. (1995).) The population characteristic of GA as well as the mutation probability makes GA well suited for this problem.

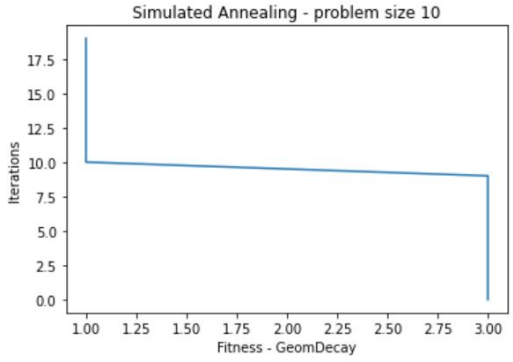


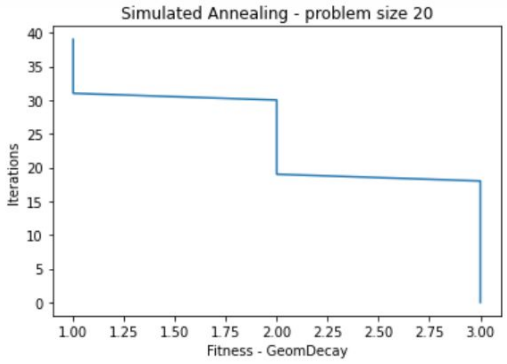
Problem 2 - Max K Color

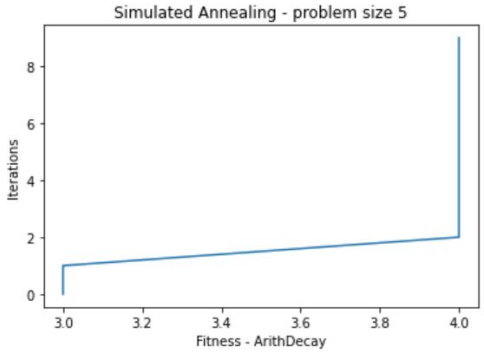
For the Max K Color problem, I gave each algorithm problem sizes of 5, 10, and 20. In all three problem sizes, GA converged to fitness of 6. RHC converged to fitness 6 at problem size 5, then converged to fitness 4 at problem size 10, then converged to fitness 3 at problem size 20. So RHC got

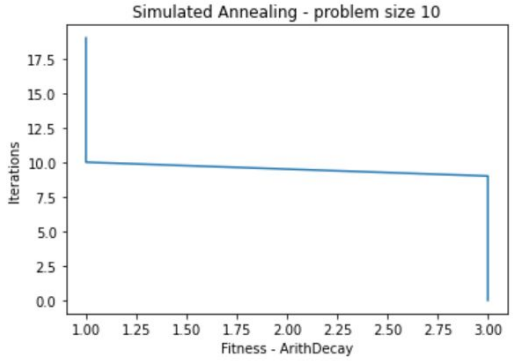
worse as the problem sizes got larger, which makes sense as it can be thought of as an algorithm for finding peaks in a graph, and if the problem size is getting larger, that can be thought of as more peaks for it to search for. SA did well at first, quickly converging to fitness 4 at problem size 5 for all types of schedule. Then SA's fitness curves became inverted for all other problem sizes. The Max K Color problem highlighted the advantages of SA because it was able to find its best fitness curves at the earliest number of iterations, and it did so faster than any of the other algorithms in terms of wall time.

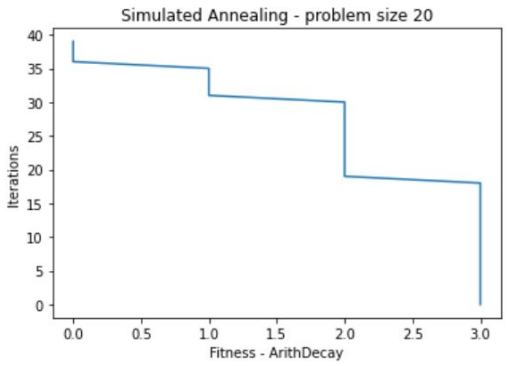


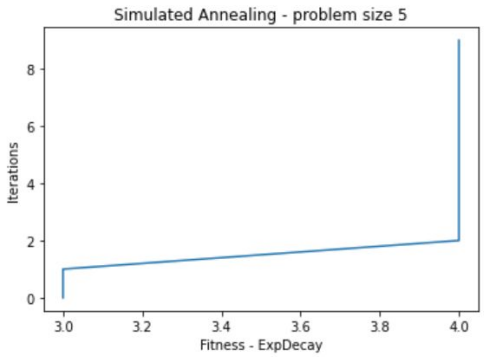


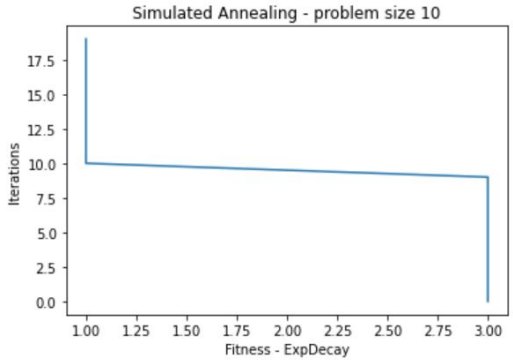


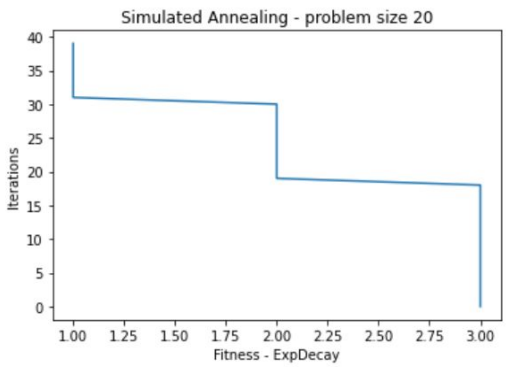






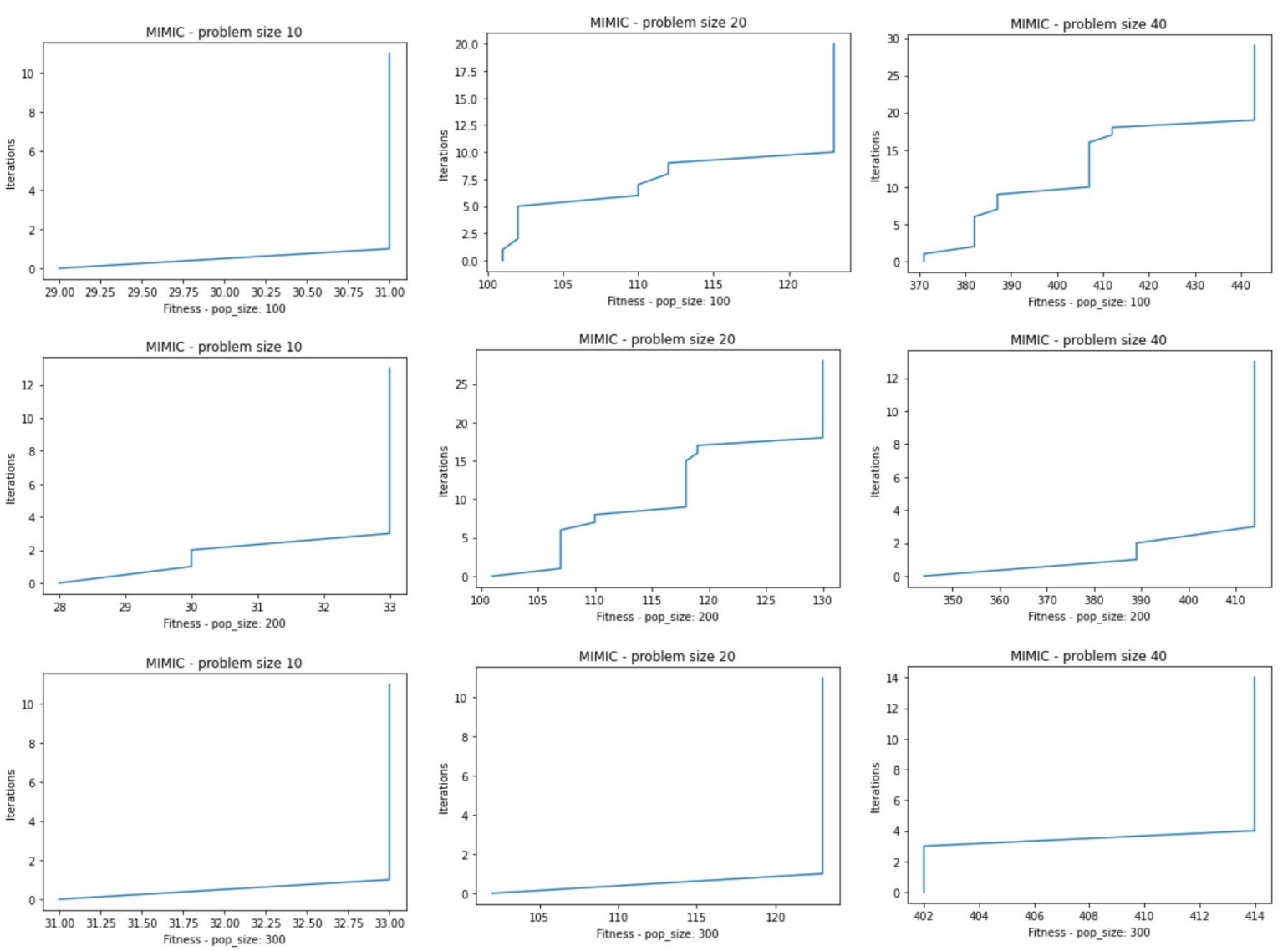






Problem 3 - Knapsack

For the Knapsack problem, I gave each algorithm problem sizes of 10, 20, and 40. RHC resulted in 0 fitness at all iterations for all values of *restarts* at all problem sizes. Similar results were found with SA: 0 fitness at all iterations for all values of *schedule*. These two clearly did the worst for this problem. This makes sense because RHC and SA can both be thought of as algorithms looking for peaks on a graph, which doesn't really apply to the Knapsack problem. At problem sizes 10, 20, and 40, GA converged on fitnesses 33, over 130, and 540, respectively. MIMIC did better by the following: at problem size 10, converging to a fitness of 33 by 1 iteration with a population of 300; at problem size 20, converging to a fitness over 120 by 1 iteration with a population of 300; and at problem size 40, converging to a fitness of 414 by the 4th iteration with a population of 300. While both MIMCIC and GA have a population size component that relates to the Knapsack problem, this problem highlighted the advantages of the MIMIC algorithm which performed in a consistently stable manner due to the way in which it performs more calculations per iteration. This can make this algorithm perform more slowly than other algorithms, for optimization problems that can require more computation like Knapsack, this is acceptable.



Part 2

For the second part of this assignment, I employed neural networks using randomized optimization algorithms to analyze one of the problems from the first assignment. The problem I chose from the first assignment was that of phonemes. (Lee, John A. (2000).) The code and results of this part can be found in "*A2 - Randomized Optimization - Part 2 - dstrube3.ipynb*".

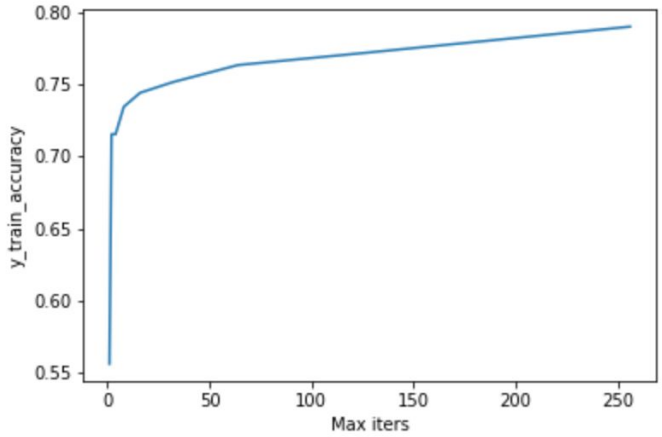
Since Assignment 1 used sklearn with unscaled data and this assignment uses mlrose with scaled data, I re-ran the neural network (NN) using the hyperparameters from Assignment 1 (*activation*=tanh,

hidden_nodes=(9,8)) in a NN using gradient descent (GD) to use as a comparison to the algorithms for this assignment. With this setup, GD looked for its best accuracy by searching through *max_iters* from 1 to 256 (multiplying by 2 for each step), and *learning_rate* from 0.01 to 10 (multiplying by 10 for each step). GD found a best accuracy of 0.7067 with *max_iters*=2 and *learning_rate*=1. This was only the second step for the first hyperparameter and the third step for the second hyperparameter. Why is that? It seems that the number of iterations for just a gradient descent has a little effect on the accuracy of the model, but not much. (If there was no effect, then the best accuracy would be found at *max_iters*=1.) The learning rate has a little more of an effect on the accuracy, but not with the maximum value of the learning rate. Why is that? Given that the learning rate can be thought of as the amount of distance traveled before taking another measurement, it indicates that the granularity of the data is fine but not too fine. This accuracy of 0.7067 is not as good as the 0.8554 found using sklearn. Why is that? One reasonable conclusion is that it's due to sklearn's NN having different hyperparameters than mlrose's. But still, I got the mlrose NN as close to the one from sklearn as I could, that's what I'll use going forward. Let's move on to exploring the assigned randomized optimization algorithms.

For each of the randomized optimization algorithms, I initially ran with a test size of 0.2, and then I ran through them again with test size of 0.1 and then 0.3. The following reported results are taking the averages of all these runs.

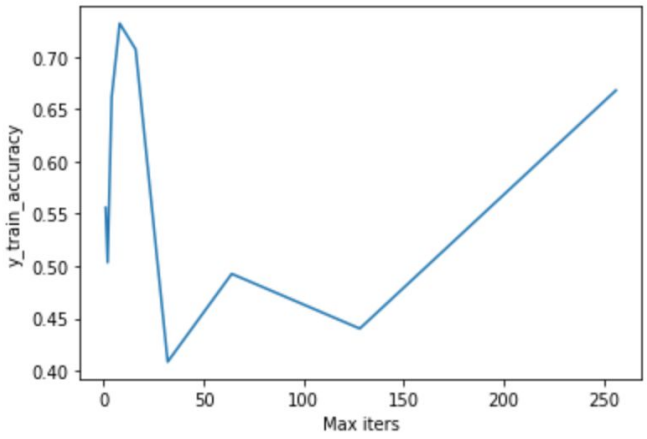
For RHC, holding *restarts* constant at 10, searching through *max_iters* and *learning_rate*, average best accuracy of 0.7816 against the training data was found at *max_iters*=256 and *learning_rate*=10. (Against test data, the average best accuracy was 0.7624.) Both of the hyperparameters *max_iters* and *learning_rate* had a significant effect on the accuracy. Searching through more values of these hyperparameters may have given better accuracy, but that may have led to overfitting. Holding *max_iters* constant at 256, searching through *restarts* and *learning_rate*, best accuracy was found at *restarts*=5, but the accuracy was not significantly changed. This indicates that *restarts* is not a significant factor in finding the best model using RHC for this data. This could be because the data was scaled, i.e., smoothed out, and the number of random restarts wouldn't have affected the results as much if there was no scaling.

The fact that RHC did better as the test size grew (and training size shrank) is puzzling. One interpretation is that RHC is not well suited for this data. Doing more runs at more varieties of training and testing sizes may prove or disprove this hypothesis. Another interpretation is that the best values of hyperparameters were not found. Doing more runs with differing ranges of hyperparameter values would help answer that question.



RHC - Max_iters compared to accuracy with test size = 0.3

For SA, holding the *schedule* constant at the default GeomDecay, average best accuracy of 0.7331 against the training data was found at *max_iters*=8 and *learning_rate*=1.0. (Against test data, the average best accuracy was 0.7159.) Both of the *max_iters* and *learning_rate* had a moderate effect on the accuracy, i.e., it wasn't at the start or end of the ranges of these that found the best accuracy. Note, however, since the range of *max_iters* wasn't linear and the chart *max_iters* and accuracy showed an upward trend after the maximum accuracy found at *max_iters*=8, this assessment may be revised upon exploring a greater range of *max_iters.*



SA - Max_iters compared to accuracy with test size = 0.3

These moderate effects may have been more significant if multiple values of *schedule* had been explored in these runs, because each possible *schedule* value has multiple parameters. So I next explored the schedule GeomDecay with the above values for *max_iters* and *learning_rate.* Holding *min_temp* constant at 0.001, average best accuracy of 0.7389 was found at *init_temp*=1 and *decay*=0.1. Holding *decay* constant at 0.1, the best accuracy of 0.7389 was found at *init_temp*=1 and *min_temp*=0.001. The fact that both cases (holding *min_temp* constant and holding *decay* constant) found the same best accuracies using the same hyperparameter values indicates that these are the optimal hyperparameter values for GeomDecay using SA on this data. The *min_temp* values used ranged from 0.001 to 1 with a multiplier of 10. I found that the best accuracies could be achieved by starting with lower and lower *min_temp* values, but there was a diminishing return once the minimum *min_temp* dropped below 0.001. The *init_temp* values ranged from 1 to 20. The decay values ranged from 0.1 to 1.0, incrementing by 0.3. So the optimal hyperparameter values were at their lowest settings. But why? It could be because the *decay* and *init_temp* didn't affect the outcome as much as the *min_temp*. The *min_temp* basically controls how fast the search goes over all the data. A lower *min_temp* means going over the data more slowly, that is, more carefully looking for the most accurate model. Similar results were found with the arithmetic decay and exponential decay schedules.

For GA, holding the *pop_size* constant at *5* and *mutation_prob* at 0.5, average best accuracy of 0.7606 against the training data was found at *max_iters*=256 and *learning_rate*=0.01. (Against test data, the average best accuracy was 0.797.) Like RHC, GA found better accuracy with more max_iters, but expanding the range of max_iters being searched through would probably have led to overfitting. Holding *max_iters* constant at 256 and *learning_rate* at 0.01, average best accuracy of 0.8019 was found at *pop_size* = 12 and *mutation_prob* = 0.01. This accuracy was greater than that found holding *pop_size* and *mutation_prob* constant at the aforementioned values. The values of *pop_size* ranged from 2 to 20, and the values of *mutation_prob* ranged from 0.0001 to 10 using a multiplier of 10. The optimal values of *pop_size* and *mutation_prob* were somewhere near the middle of both ranges. Why is that? It's possible that this is just the nature of the data that makes these hyperparameter values yield this accuracy.

Since the accuracy found by GA was greater than the accuracy found by any of the other randomized optimization algorithms in this part of this assignment, it seems that GA was best for finding the best accuracy for a model using this data. However, it is worth noting that the randomized optimization algorithms that did better also took longer: GA took an average of 20 minutes and 31 seconds. RHC took an average of 9 minutes and 12 seconds. SA took an average of 1 minute and 15 seconds. This makes sense because GA had to take into account population sizes and mutation probabilities at each iteration, RHC had to take into account the number of restarts, whereas SA (even with its more customizable parameters of schedules and each schedule's parameters) only had to deal with the initial temperature and the minimum temperature of each decay schedule's iteration.

References

Hayes, G. (2019). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python. [https://github.com/gkhayes/mlrose]. Accessed: 2020-10-11.

Rollings, A. (2020). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python, hiive extended remix. [https://github.com/hiive/mlrose]. Accessed: 2020-10-11.

Pedregosa et al. (2011). JMLR 12, pp. 2825-2830, Scikit-learn: Machine Learning in Python. [http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html].

Lee, John A. (2000). The ELENA Project [https://www.elen.ucl.ac.be/neural-nets/Research/Projects/ELENA/elena.htm]. Louvain-la-Neuve, Belgium: Université catholique de Louvain, Neural Network Group

Baluja, S. and Caruana, R. (1995). Removing the genetics from the standard genetic algorithm. Technical report, Carnegie Mellon Univerisity.