CS7641 Machine Learning Fall 2021
Project 2 : Randomized Optimization

Ruhan Li
(rli445)

Abstract

Four randomized optimization algorithms (Randomized Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithm (GA)) were implemented to three classic optimization problem domains (Max K-Color, Four Peaks, and Flip Flop). This report has two parts. The first part focuses on finding the best algorithm for each problem domain, and the second part talks about finding good weights of Neural Network structure that I found in Assignment 1 using the randomized optimization algorithms. The implement of algorithms and curves are based on python and with public libraries, including sklearn, mlrose_hiive, and pandas.
Link of code: https://github.com/Hanlarious/ML/tree/main/A2

Part I: Random Search Algorithms

In this section, four local random search algorithms, namely Randomized Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithm (GA), and MIMIC, are implemented to find optimized solutions for three problems, including Max K-Color problem, Four Peaks problem, and Flip Flop problem.

General Implement

For all three problem domains and four algorithms, I followed the following progress:

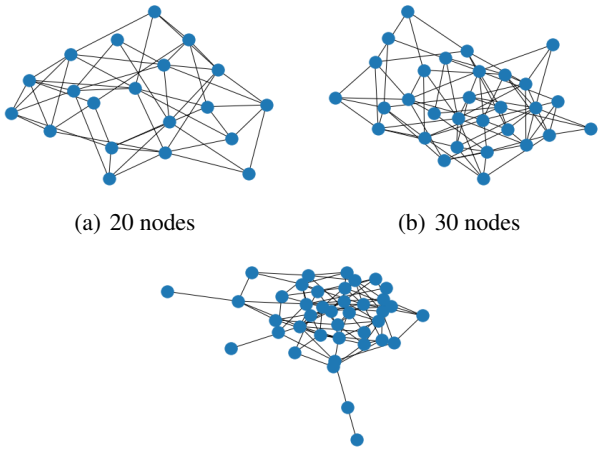
1. **Try different problem sizes:** For each problem, I generated three different problem sizes, representing small, medium, and large sizes, to observe the performances of each algorithm on different scale of problems. While running the same problem with the same algorithm and same size multiple times, I found that the fitness curves varied largely each time. However, no matter how the shape changed, the basic rules were the same: almost all the algorithms constantly performed better on a certain scale of problem size.
2. **Tune parameters:** I first set the *max_iter* to a fairly high number (1000), because based on the operation mechanism of *mlrose_hiive*, it stops running when convergence happened[1]. Therefore, I will not face the problem of stopping iteration before convergence because the small number of *max_iter* has reached. With this as my base parameter, I tuned other important parameters for each algorithm, including *maximum attempts*, *decay schedule*, *population size*, *mutation probability*, and *keep percentage*. In order to control the variables, I only tuned one parameter at a time, and kept that value with best performance fixed when searching for next value of parameter.
3. **Run with different seeds and average the results:** Because of the huge randomness of these algorithms, sometimes we could easily find the optimized solution with good luck but sometimes we may get stuck just because of a bad starting point. Therefore, for each algorithm with tuned parameters, I ran 5 times with different seeds and averaged the results.
   In order to be as fair as possible, I used the same set of seeds for each algorithm, ranging from 0 to 88888888 (I was hoping that the huge variance of seed number can cover more possibilities). For the same reason, I canceled the parameter of *restart*. Since the aim was to find the best algorithm, I felt like I needed to minimize the potential bias, nevertheless *restart* brought more uncontrollable factors.
4. **Compare *best fitness*, *wall time*, and *number of iterations to converge*:** The previous tuning process promised the best performance of each algorithm, which made the comparison between each algorithm convincing. I will define my understanding of "best" in detail in the analysis section.

Max K-Color problem

This problem is to minimize the number of pairs of adjacent nodes with the same color. Therefore, I made my results time *-1* to make the fitness curves go up. The values on the y-axis did not match the real fitness scores, but the behaviors and trends made more sense in this problem.

- **Different sizes:** I generated the problem with *max_connections_per_node* of 4, and *max_colors* of 3, and set the number of nodes to 20 (Figure 1(a)), 30 (Figure 1(b)), and 40 (Figure 1(c)).

[1] I drew this conclusion by observation. I noticed that when trying out performances on different problem sizes, the iteration stopped at different numbers even if the provided number of *max_iter* was the same. Even thought I failed to find documentation to prove my inference, I believer that *mlrose_hiive* somehow handled the iteration stopping criteria for us.

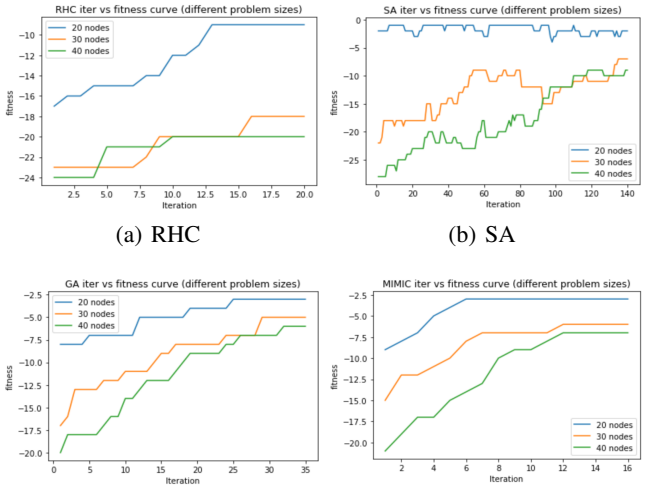


(a) 20 nodes (b) 30 nodes

(c) 40 nodes

Figure 1: Max K-Color problem

For this problem, the performance of four algorithms are similar, except that RHC had relatively worse performance with problem size of 30 and 40 nodes (Figure 2(a)), whereas the problem with 20 nodes had a stable and slightly better performance on this SA (Figure 2(b)). Since I needed to find out the "best" algorithm, I wanted to choose the problem size that most algorithms can perform best. In this case, it will be 20 nodes.



(a) RHC (b) SA

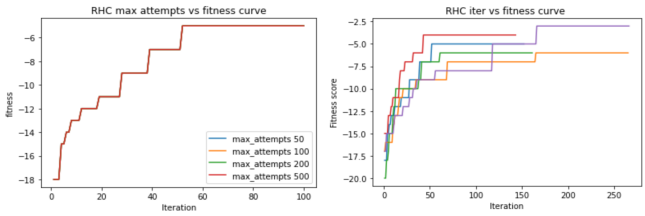
(c) GA (d) MIMIC

Figure 2: fitness on different problem sizes

Different algorithms

– **RHC**

∗ *average converge time: 0.014s*
∗ *average fitness score = 4.8*
∗ *approximate number of iterations to converge = 220*



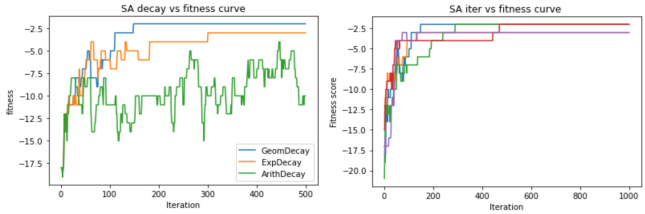
(a) tuning:maximum attempts (b) fitness: 5 seeds

Figure 3: RHC parameter tuning and model performance

The RHC algorithm did not have many parameters to tune, so in the end, I only had two parameters fixed: *maximum attempts = 200*, and *maximum iterations = 1000*. I then calculated the best fitness and recorded the wall time for each seed. The numbers showing above is the averaged results.
The tuning curve (Figure 3(a)) suggests that no matter what the *maximum attempts* is, the fitness is the same. In fact, I found that other algorithms and other problems had the same performance in the tuning of *maximum attempts*: the value of *maximum attempts* would not affect the fitness. Therefore, I will set *maximum attempts* to 200 in the rest of my project without further explanation.
The fitness curve (Figure 3(b)) shows that the curves of 5 seeds are similar with fitness increase while iteration increases, but their convergent times are different.

– **SA**

∗ *average converge time: 0.08s*
∗ *average fitness score = 2.4*
∗ *approximate number of iterations to converge = 700*



(a) tuning:decay schedule (b) fitness: 5 seeds

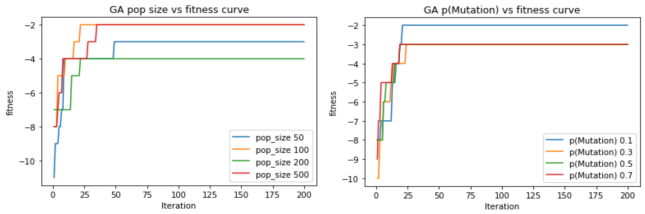
Figure 4: SA parameter tuning and model performance

For SA, I tuned the *decay schedule* (Figure 4(a)), which decided how the temperature varied for each iteration.[2] It looks that GeomDecay and ExpDecay have comparable performance whereas GeomDecay turns out slightly better.
The fitness curve (Figure 4(b)) shows that the fitness can increase fast but the convergence may take long if a unlucky seed was chosen.

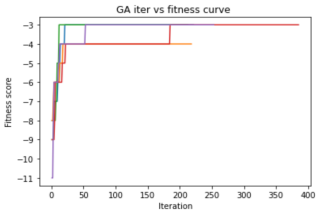
– **GA**

∗ *average converge time: 2.8s*
∗ *average fitness score = 3.2*

- *approximate number of iterations to converge = 350*



(a) tuning:population size (b) tuning:mutation probability



(c) fitness: 5 seeds

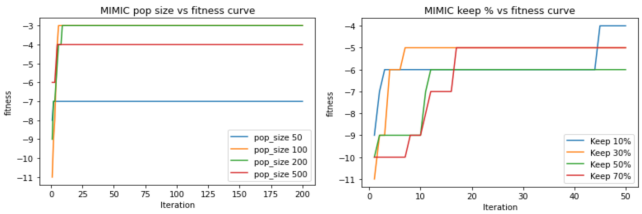
Figure 5: GA parameter tuning and model performance

I tuned the *population size* (Figure 5(a)) and *mutation probability* (Figure 5(b)) here. *population size* controls the population size being used in the algorithm, and *mutation probability* represents the probability of mutations for reproduction. [2]
According to the tuning curves, *mutation probability* = 0.1 is no doubt the winner. For *population size*, even though population size 500 reached the same fitness score as population size 100 did, it took more iterations to converge. So I finally decided to choose the population size of 100 and the probability of mutation of 0.1.
The fitness curve (Figure 5(c)) shows that even though three out of five runs converged fast, there were two trials that converged after 100 to 250 iterations. This makes me feel like the randomness of the GA algorithm is big.

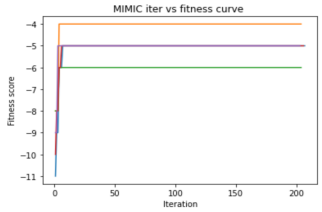
– **MIMIC**

- *average converge time: 40.42s*
- *average fitness score = 5*
- *approximate number of iterations to converge = 20*

I tuned the *population size* (Figure 6(a)) and *keep percentage* (Figure 6(b)) here. *population size* works the same as in the GA algorithm, and *keep percentage* controls the ”proportion of samples to keep at each iteration”.[2]
We can tell from the tuning curve of *population size* that both size 100 and size 200 reached the highest fitness score within 10 iterations, in which population size 100 was slightly faster, therefore, I decided to set it to 100.
On the other hand, even though 10% worked the best for percentage to keep after 40 iterations, in my point of view, it is common for the MIMIC algorithm to converge before 40 iterations. Therefore, I think the performances in the smaller iterations are more important.



(a) tuning:population size (b) tuning:keep percentage

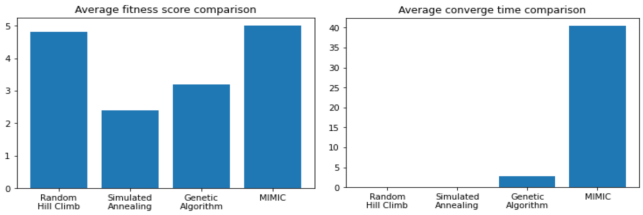


(c) fitness: 5 seeds

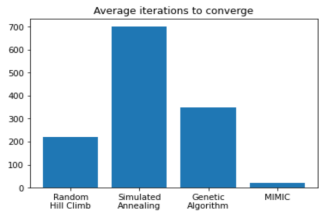
Figure 6: MIMIC parameter tuning and model performance

Therefore, I decided to set the *keep percentage* to 30%. The fitness curve (Figure 6(c)) shows that all five runs converged with a really small number of iterations with high fitness scores, **which highlighted the advantages of the MIMIC algorithm: convergence with few iterations, as well as high fitness**.

– **Comparison**



(a) average fitness (b) average converge time



(c) converge iterations

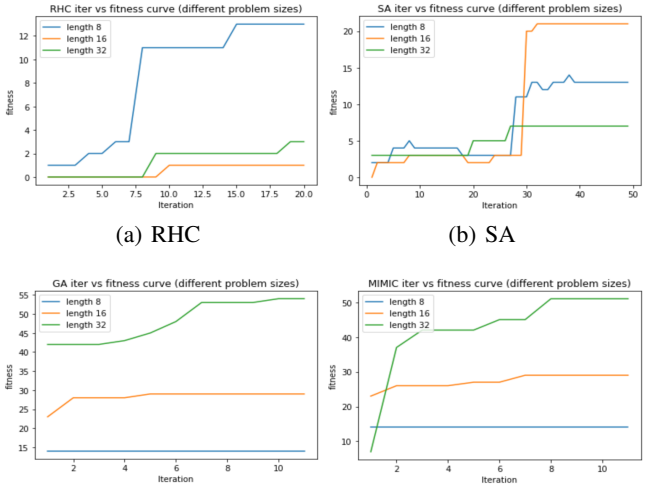
Figure 7: Comparison of algorithms

From the comparison graphs, it is clear that the algorithm with the highest average fitness score is the MIMIC algorithm (Figure 7(a)). Even though it takes longer time to execute (Figure 7(b)), the iterations it takes to converge has a huge advantage over others (Figure 7(c)). In the situation where the cost of every iteration is high, the MIMIC algorithm can save plenty of expenses.

Four Peaks problem

This problem is to find the maximum difference between the number of consecutive 0s and the number of consecutive 1s in given bit strings.

- **Different sizes:** I generated the problem with length of 8, 16, and 32 respectively.



(a) RHC (b) SA

(c) GA (d) MIMIC

Figure 8: fitness on different problem sizes

For this problem, RHC had the best performance on the smallest problem size (Figure 8(a)), where as other algorithms performed better on bigger problem sizes. Especially GA and MIMIC had clearly higher fitness scores on the largest problem size (Figure 8(c) and (d)).
What's interesting here is that for the bit length of 8, GA and MIMIC constantly returned me the same fitness score with a horizontal line, regardless of the number of times that I re-ran the algorithms. I felt like the algorithms somehow got stuck and could not find an optimized solution, meaning that these two algorithms could not solve the Four Peaks problem with 8 bits input. Due to the reasons stated above, I decided to set the problem with the bit length of 32.

Different algorithms

– **RHC**

∗ *average converge time: 0.022s*
∗ *average fitness score = 32.0*
∗ *approximate number of iterations to converge = 700*

The same as the *Max K-Color problem*, the tuning of *maximum attempts* did not help improving the performance. So I'm not wasting space to show its curve.
Moreover, because the logic of tuning parameters, as well as the parameters that had been tuned for each algorithm were the same as in the *Max K-Color problem*, I will not discuss them in detail again unless I found noticeable behavior.
I set *maximum attempts = 200*, and *maximum iterations = 1000*, and got the fitness curve (Figure 9). It looks that the the performance of RHC isn't good, considering its low fitness score and the large number of iterations for convergence.

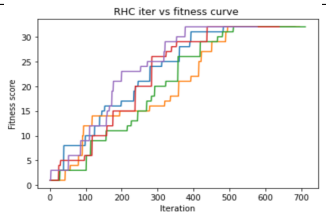
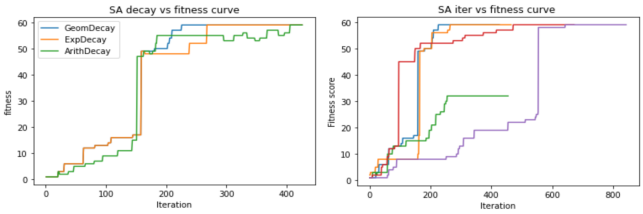


Figure 9: RHC model performance

– **SA**

∗ *average converge time: 0.023s*
∗ *average fitness score = 53.6*
∗ *approximate number of iterations to converge = 800*



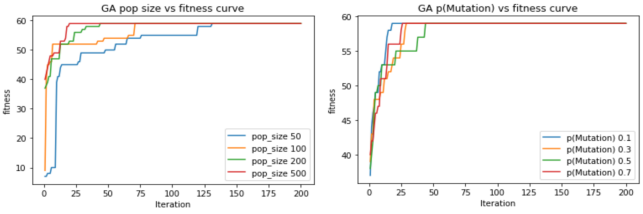
(a) tuning:decay schedule (b) fitness: 5 seeds

Figure 10: SA parameter tuning and model performance

For SA, I tuned the *decay schedule* (Figure 10(a)). It looks like the GeomDecay reached the highest fitness score within the fewest iterations.
The fitness curve (Figure 10(b)) shows huge variance on convergence, which meets the characteristics of the SA algorithm. Because the move with worse fitness scores can be accepted as the next step if the temperature is high,[1] it is possible that the algorithm takes extra iterations to converge with a bad luck.

– **GA**

∗ *average converge time: 9.35s*
∗ *average fitness score = 59.0*
∗ *approximate number of iterations to converge = 70*



(a) tuning:population size (b) tuning:mutation probability

Figure 11: GA parameter tuning and model performance

It is obvious from the tuning curves that *population size = 500* (Figure 11(a)) and *mutation probability = 0.1* (Figure 11(b)) are the best values.

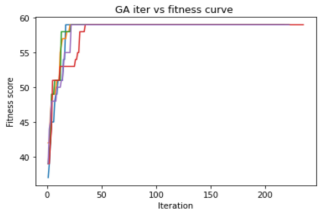


Figure 12: fitness: 5 seeds

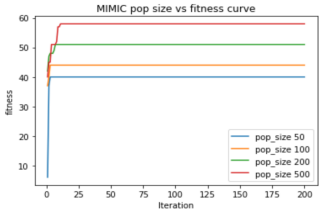
The fitness curve (Figure 12) shows great performance of the GA algorithm here. All five runs converged with few iterations and a high fitness score.

– **MIMIC**

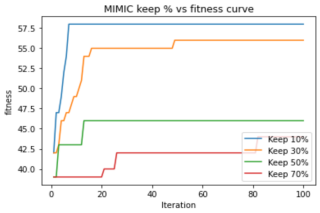
∗ *average converge time: 109.9s*

∗ *average fitness score = 57.6*

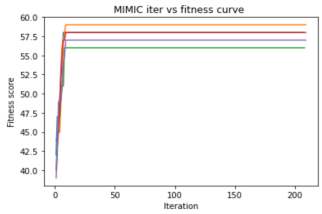
∗ *approximate number of iterations to converge = 20*



(a) tuning:population size



(b) tuning:keep percentage



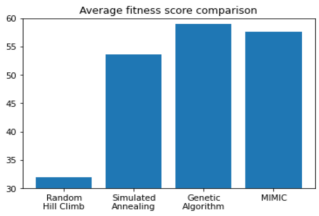
(c) fitness: 5 seeds

Figure 13: MIMIC parameter tuning and model performance

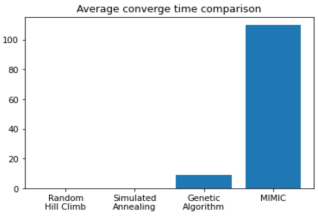
Based on the tuning curves, the *population size* should be set to 500 (Figure 13(a)) and the *keep percentage* needs to be set to 10% (Figure 13(b)).
The fitness curve (Figure 13(c)) suggests good convergences with few iterations.

– **Comparison**

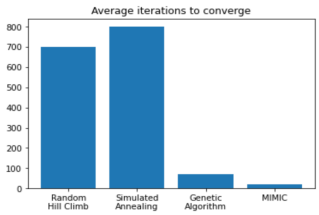
Based on the comparison graphs(Figure 14), the GA algorithm is definitely a winner! It has the highest fitness score with a fairly low converge time, as well as few iterations to converge, whereas the MIMIC algorithm has a slightly lower fitness score but takes nearly 12 times longer to converge.



(a) average fitness



(b) average converge time



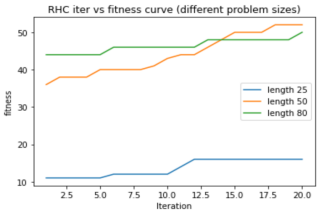
(c) converge iterations

Figure 14: Comparison of algorithms

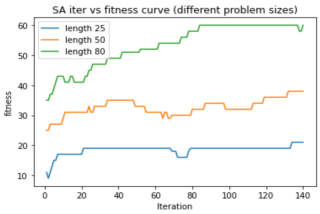
Flip Flop problem

This problem ”evaluates the fitness of a state vector x as the total number of pairs of consecutive elements of x”.[3]

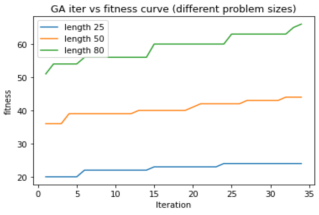
- **Different sizes:** I generated the problem with length of 25, 50, and 80 respectively.



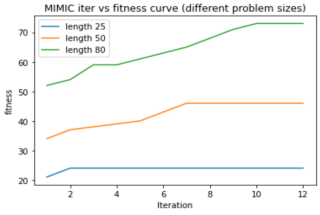
(a) RHC



(b) SA



(c) GA



(d) MIMIC

Figure 15: fitness on different problem sizes

For this problem, all four algorithms had really similar behaviours. It seems that they all performed significantly better on bigger problem sizes. The only difference was that RHC algorithm did not distinguish the performances on bit length 50 and 80 clearly as the other three algorithms did, but because 3/4 of the algorithms performed the best with the largest problem size, I still decided to generate the problem with the size of 80 bits.

Different algorithms

- **RHC**
  - *average converge time: 0.087s*
  - *average fitness score = 64.6*
  - *approximate number of iterations to converge = 350*

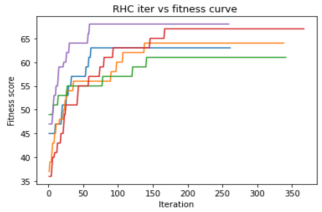
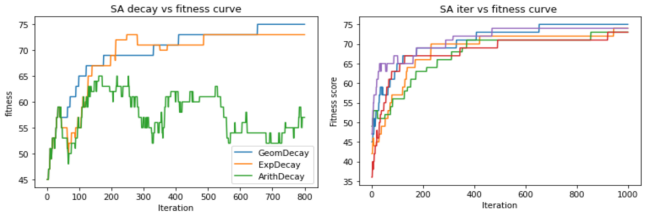


Figure 16: RHC model performance

With *maximum attempts = 200*, and *maximum iterations = 1000*, the convergence of the RHC algorithm seems to vary largely with different seeds.

- **SA**
  - *average converge time: 0.32s*
  - *average fitness score = 73.8*
  - *approximate number of iterations to converge = 1000*



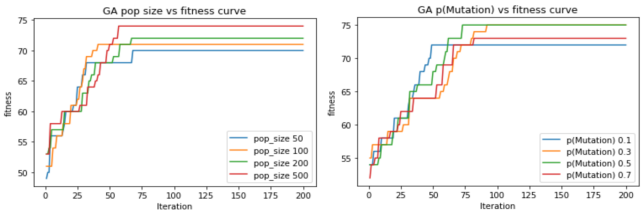
(a) tuning:decay schedule (b) fitness: 5 seeds

Figure 17: SA parameter tuning and model performance

For SA, I tuned the *decay schedule* (Figure 17(a)). It looks like the GeomDecay and ExpDecay had similar performance, whereas AirthDecay did not perform well. Because SA algorithm normally takes more iterations to converge, compared with other algorithms, I feel like the fitness with larger iteration numbers are more important. Therefore, I decided to choose the GeomDecay, which performed better after 700 iterations. The fitness curve (Figure 17(b)) shows that all five runs gradually converged to a high fitness score.

- **GA**
  - *average converge time: 40s*
  - *average fitness score = 72.8*
  - *approximate number of iterations to converge = 350*

It is obvious from the tuning curves that *population size = 500* (Figure 18(a)) is the best value for population size. For the *mutation probability* (Figure 18(b)), it seems that the value 0.5 had converged to the highest fitness score first.



(a) tuning:population size (b) tuning:mutation probability

Figure 18: GA parameter tuning and model performance

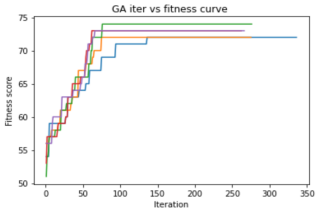
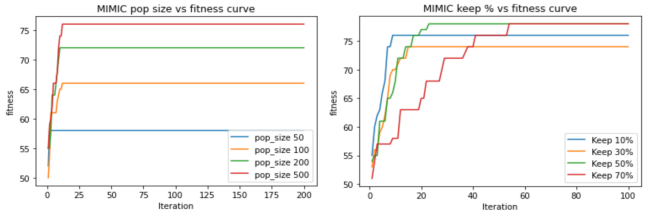


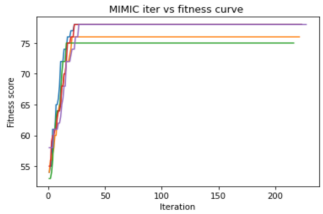
Figure 19: fitness: 5 seeds

The fitness curve (Figure 19) shows that 4/5 of the different runs converged fast, whereas one of them converged slightly slower with a lower fitness score.

- **MIMIC**
  - *average converge time: 741.14s*
  - *average fitness score = 77.0*
  - *approximate number of iterations to converge = 100*



(a) tuning:population size (b) tuning:keep percentage



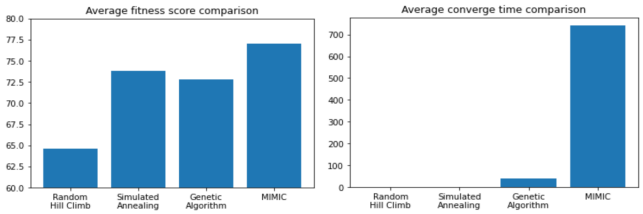
(c) fitness: 5 seeds

Figure 20: MIMIC parameter tuning and model performance

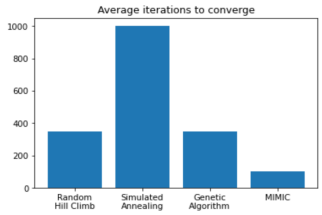
Based on the tuning curves, it is obvious that *population size = 500* beat other candidates (Figure 20(a)). For the *keep percentage*, the value of 50% converged to the highest fitness score with a small number of iterations, so I will choose it as the percentage to keep (Figure 20(b)).

The fitness curve (Figure 20(c)) suggests good convergences with a small number of iterations.

– **Comparison**



(a) average fitness (b) average converge time



(c) converge iterations

Figure 21: Comparison of algorithms

Despite the fact that MIMIC has the highest average fitness score (Figure 21(a)), I would not think it is the best algorithm for this problem, since it took too long to converge. I would probably say SA, with the second highest fitness score is the best model, even though it takes more iterations to converge(Figure 21(c)). The wall clock time performance of SA algorithm is way better than the MIMIC and GA algorithm (Figure 21(b)). Therefore, in the case of high time requirements, SA algorithm is the best option for this problem.

Part II: Neural Network Optimization

In this section, three randomized optimization algorithms (RHC, SA, GA, and MIMIC) were used to find good weights for the Neural Network of my Assignment 1.

Data set: Phishing Websites

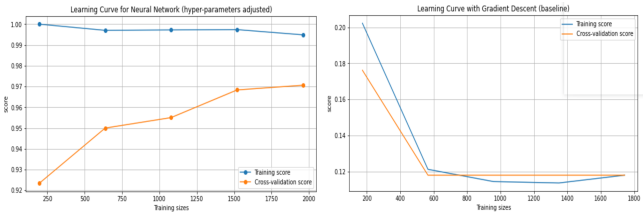
I'm using one of the data sets that I used for Assignment 1. The data set has 2456 entries with 31 columns, and no missing data.

Preparation

I did the same preparation work as I did for Assignment 1:

- Make data set balanced using *Over-sampling* technique
- Split data into *training set* and *test set*
- Scale data

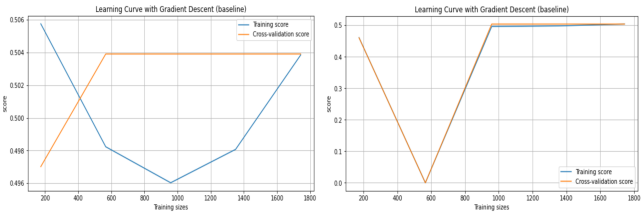
To make sure I'm comparing apples to apples, I ran the *MPLClassifier* again, and got the same accuracy score as before.
Unfortunately, the best activation function I had before was "logistic", which was not available in the *mlrose* library. I had to substitute it with "tanh", which had very similar performance with "logistic" according to the validation curve I plotted for Assignment 1. The model performed pretty good using *sklearn* library, returning an accuracy score of 0.96 (Figure 22(a)).



(a) sklearn (b) mlrose

Figure 22: Learning Curve

I brought the same structure to the NeuralNetwork function provided by *mlrose* library, and set the "algorithm" to "gradient_descent", suprisely, I had an extremely low accuracy (Figure 22(b)).
To figure out what happened, I did some research. It seems that even though both libraries uses the gradient descent method, the *sklearn* library uses Adam gradient descent, which is an advanced version.
Because of the extremely poor performance within *mlrose* library, I had to tune the parameters again, wishing to find a break through. I changed the activation function first, and found that the accuracy increased a little bit, but still swing around 0.3. Then I started to think that maybe my learning rate was set too high, so the algorithm kept missing the global maxima? So I tried different small learning rate, ranging from 0.000001 to 0.01. However, this did not help that much. Then I started to doubt the hidden layers of my original structure. Originally I had one layer with 110 nodes, I felt like this might be too much for a data set with only 2000 data points. So I started to reduce it to two layers with 10 nodes in each layer.
After hours of trying, I found that I got exactly the same cross validation accuracy score with very different structures. So I plotted out the learning curves for two different structures with the same accuracy score to observe the behaviors (Figure 23).



(a) gradient decent model1 (b) gradient decent model2

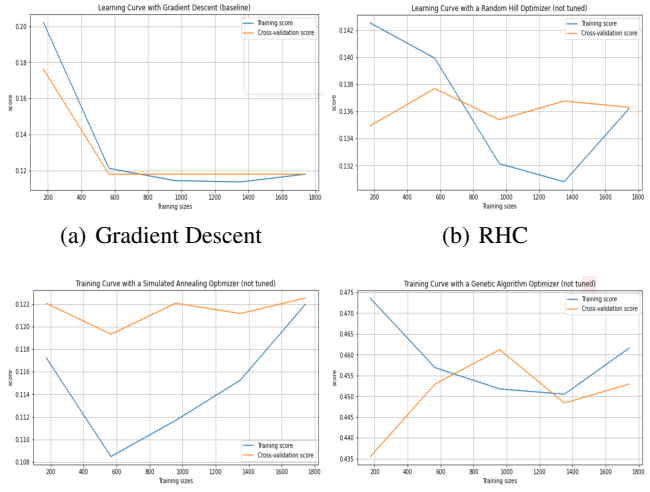
Figure 23: Learning Curves

It is obvious that the behaviours of these two structures

were very different, even though they had the same accuracy score. It seems to me that none of them converged. Because the score was the best accuracy score that I got. I had to use the model as my baseline gradient descent model to tune and compare with other algorithms. It is not a good start, but I had to accept it.
I've tried my best, but I could not figure out what happened. I guess it might be that I have chosen a bad data set, which made the algorithm stuck somewhere. So if I had more time, I may change the data set to see if I can improve the performance.

Apply randomized optimization algorithms

Because the performance with original structure was too poor, I changed the structure of NN, therefore, I will return two sets of curves: one without changing the NN structure, and one with tuned NN structure.
To make the comparison meaningful, I used the new structure that returned the best accuracy score in the gradient descent. Moreover I found that the random seeds had huge impact on the result. Originally, I planed to set a fix list of random seeds, just like what I did for Part I, and average the results, but I ran out of time, so I tried around 30 different seeds in the gradient descent, and picked the one with the best performance, and kept it fixed when tuning parameters of other algorithms.
On top of the NN structure, I tuned other parameters for each algorithm. With the same theory that I mentioned before, I gave a fairly big number to *maximum iteration*, and set the *early stopping* parameter to *True*, so that the iteration would not be interrupted before convergence.
I mainly tuned the *learning rate*, *maximum attempts*, *population size*, *mutation probability*, and *restarts*.



(a) Gradient Descent (b) RHC

(c) SA (d) GA

Figure 24: Learning Curves

These are the learning curves for the randomized optimization algorithms with the original NN structure. Base on the curves (Figure 24), it is easy to generate the conclusion that none of the algorithms converged. I think it is not the issue that the iteration stopped before convergence. The problem should be the NN structure or the data set itself.
I also recorded the accuracy scores after 5-fold cross validation for each algorithm (Table 1). It is sad but interesting to see that gradient descent actually had the lowest accuracy, whereas the accuracy of GA is four times that of other algorithms.

Algo	GD	RHC	SA	GA
Accuracy	0.12	0.14	0.12	0.45

Table 1: Accuracy score after 5-fold cv

I don't know what exactly was the reason, but as I analysed before, since changing the NN structure improved the performance, I'm still convinced that because of the different ways of implementing the gradient descent, the NN structure that performed well with *sklearn* library somehow lost its advantage and performed particularly poor in gradient descent.
It is possible that my previous model had the problem of over-fitting. But to me, the learning curve (Figure 22(a)) suggests convergence, and it looks like a good model with low bias and a little bit high variance. I don't believe that the over-fitting issue could cause such a dramatic change.

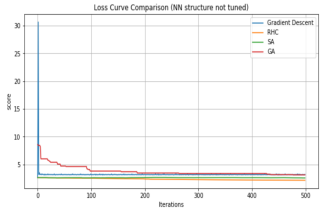
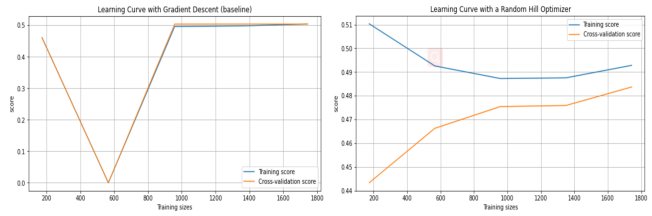
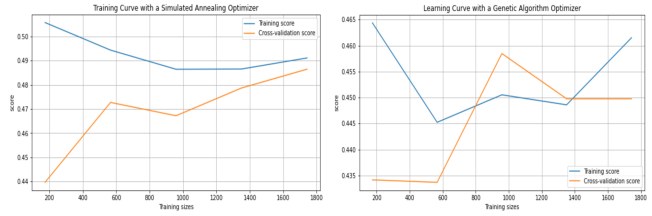


Figure 25: Loss curve comparison

The information I got from the loss curves (Figure 25) is that as expected, GA algorithm had the most "normal" behavior. It slowly went down to a relatively low and stable situation. For the Gradient Descent, it went to a very high value then suddenly dropped to a low value.
As for the reason, I'm thinking that Neural Network has the feature that gradient descent always starts with small values for weights, therefore, the starting point can be largely varied. The thing is, even if they start at a point with high loss value, it should go down later. This is like an "accident", but I think the general trend of going down is more important.
As for the loss curves of SA and RHC, I feel like they are like horizontal lines.
The terrible situation was improved by changing the NN structure. Although they still performed no better than random guess, they have improved a lot compared to their previous performance.
The learning curves (Figure 26) shows that even though the average accuracy is low, SA and RHC has the trend to converge.
To me, Gradient Decent (Figure 26(a)) shows unconvergence, high bias and extremely low variance. This



(a) Gradient Descent (b) RHC



(c) SA (d) GA

Figure 26: Learning Curves

Algo	GD	RHC	SA	GA
Accuracy	0.504	0.484	0.48	0.45

Table 2: Accuracy score after 5-fold cv

might be the case of under-fitting. RHC (Figure 26(b)) shows good trend of convergence with high bias and relatively low variance, I think this is also the case of under-fitting. SA (Figure 26(c)) is likely to be the curve with the best performance among all four curves. It shows more obvious trend of convergence, which made me believe that once I have more data points, it will converge soon.
I had high expectations on the GA algorithm, considering its good performance with original NN structure. I spent a lot of time tuning it, but I could not get a better result. I don't even know how to interpret its learning curve (Figure 26(d)). I have tried to tune the population size and mutation probability, but I could not improve the performance any further.
After tuning the NN structure, the performance of gradient descent became the algorithm with slightly better performance (Table 2), which makes sense. And the RHC and SA algorithms also performed much better than before.

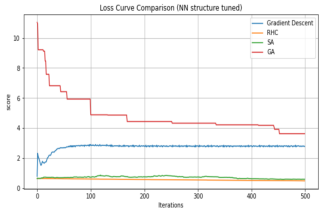


Figure 27: Loss curve comparison

However, the loss curves (Figure 27) do not agree with my previous conclusion. It looks like the loss curve of GA performed the best since it shows the trend of gradual decrease. And the loss function of Gradient Descent, which I thought had the best performance shows an increasing loss curve. For the other two algorithms I thought had largely improved their performance, still show flat lines of loss curve.
When I looked at the curves clearly, I found them somehow made sense again. Even though the GA loss curve shows a decreasing trend, it did not decrease to a low value, and for the SA algorithm, I think it is understandable for it to show some slight ups and downs since it sometimes takes the steps with a worse value.

Conclusion

The Part II of this assignment is a mess. Honestly, I'm not satisfied with the results, even thought I put a lot of effort on it. Again, it might be caused by my data set, I would love to try another data set to compare if I have enough time. However, I'm still happy to see the improved accuracy scores with tuning. The most important thing I learned from this project is that looking at one side of an object is not enough. For example, if I did not plot the learning curves for different NN structures with the same accuracy score, I would not imagine how different they behaved; if I did not plot the loss curves, I would still believe that the model with the highest accuracy score was the best.
Overall, it was a great learning experience.

References

[1] Dragan Aleksendrić and Pierpaolo Carlone. "5 - Composite materials – modelling, prediction and optimization". In: *Soft Computing in the Design and Manufacturing of Composite Materials*. Ed. by Dragan Aleksendrić and Pierpaolo Carlone. Oxford: Woodhead Publishing, 2015, pp. 61–289. ISBN: 978-1-78242-179-5. DOI: https://doi.org/10.1533/9781782421801.61. URL: https://www.sciencedirect.com/science/article/pii/B9781782421795500055.

[2] G Hayes. *mlrose: Machine Learning, Randomized Optimization and SEarch package for Python*. mlrose.readthedocs.io/en/stable/source/algorithms.html#module-mlrose.algorithms. Accessed: day month year. 2019.

[3] G Hayes. *mlrose: Machine Learning, Randomized Optimization and SEarch package for Python*. mlrose.readthedocs.io/en/stable/source/fitness. Accessed: day month year. 2019.