Assignment 2: Randomized Optimization

CHUN Hiu Sang, hchun31@gatech.edu
CS4641 Spring 2019, Georgia Institute of Technology
(Dated: March 2, 2019)

Credits:
This assignment uses mlrose to implement randomised optimisation experiments.
Hayes, G. (2019). mlrose: Machine Learning, Randomized Optimization and SEarch package for Python. https://github.com/gkhayes/mlrose.
And Prof. Charles Isbell's MIMIC tutorial.
Isbell, C.L. (2006). Randomized Local Search as Successive Estimation of Probability Densities.

I. WEIGHT OPTIMISATION FOR NEURAL NETWORK

The spotify song attributes dataset as Classification Problem 1 of my Assignment 1 is reused in this part, to find the optimised weight for a neural network model. The dataset specification is seen below:

Problem: Given list of attributes-carrying songs on spotify, labelled with sentiment ('like' or 'dislike') of a given person, predict the sentiment of that person towards a newly seen song.

Description of Dataset:

- Number of attributes: 13
- Number of instances: 2017
- Output: binary
- Source: Kaggle

Hyperparameter optimisation from Assignment 1 chooses

- num_hidden_layers = 5
- batch_size = 112

with fixed model settings

- Size of each hidden layer: 100
- Solver: Adam
- Activation: ReLu

In Assignment 1, the best performing model for this spotify dataset was Decision Trees and Boosting, not Neural Networks. The final test-time accuracy was slightly above 50%. Therefore in this assignment, we investigate how alternatives to the optimiser ma improve performance on the dataset. The problem is adjusted slightly as specified below.

The standard procedure to preprocess data is used, with a 80-20 train-test-set split, and one-hot encoding of the target train and test sets.

Adjustments made for Assignment 2:

Obviously since the solver (optimiser) is no longer a stochastic gradient variant, `batch_size` is no longer applicable here. In this Assignment, the following hypermeters would then be fixed

- num_hidden_layers = 5
- Size of each hidden layer: 10
- Activation: ReLu
- max_attempts = 100 : Maximum number of attempts in each iteration to find an improved neighbouring point
- clip_max : Clipped magnitude of weights

Note that to alleviate the issue of weight divergence (the magnitude of weights grows too large to give `NaN` values), the range of the weights are clipped by `[-1*clip_max, clip_max]` where:

- RHC: clip_max = 3
- SA, GA, MIMIC: clip_max = 1

And these three will be varied

- Solver: Randomised Hill Climbing (RHC), Simulated Annealing (SA) or Genetic Algorithm (GA)
- max_iter: Maximum number of iterations of the algorithm
- learning_rate: Learning rate

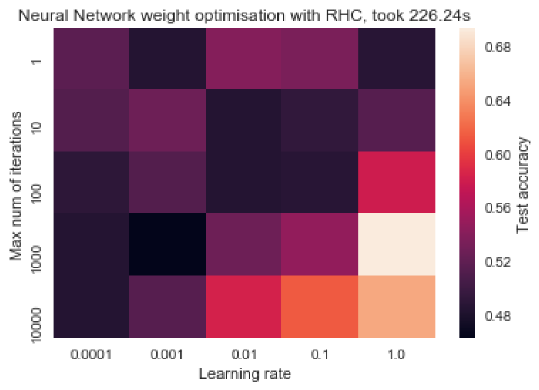
In order to see the performance of each randomised optimisation algorithm measured by test accuracy, and under which setting of `max_iter` and `learning_rate` would each of the algorithm gives the best result.

A. Randomised Hill Climbing (RHC)

Owing to limitations of the `mlrose` library, random restarts were by default set to 0 so this hyperparameter is not explored. Yet as we shall see, this algorithm performs reasonably well already.

Most performant settings:

- `learning_rate` = 1.0



- `max_iter` = 1000
- gives 69% test accuracy
- overall takes 226 seconds

This test-accuracy heatmap shows a landscape of using RHC to optimised neural network weights by various learning rates and maximum number of iterations. The patches of purple hues at the top left half of the charts indicate that RHC performs poorly on a low learning rate, with a justification that since learning rate represents step size, a step too small would not help the climber escape from a suboptimal. The fact that the leftmost two columns with learning rate below $lr \leq 0.001$ are consistently dark in colour, reveals that increasing the number of iterations would not help cope with the situation.

Therefore with a higher learning rate, optimally $lr = 1.0$, RHC by `mlrose` performs well, with best accuracy at 69%. This is quite an impressive result, considering backpropagation by `sklearn` gives around 50% of accuracy.

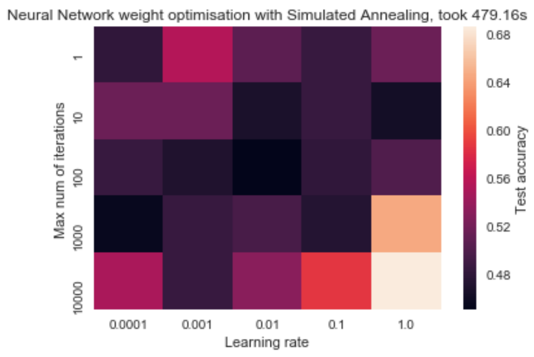
The maximum number of iterations is best at `max_iter = 1000`, and beyond that point, test accuracy starts to fall. This can be explained by balancing exploration and exploitation, that having too many iterations could allow the climber to fall off from the optimum and continue searching.

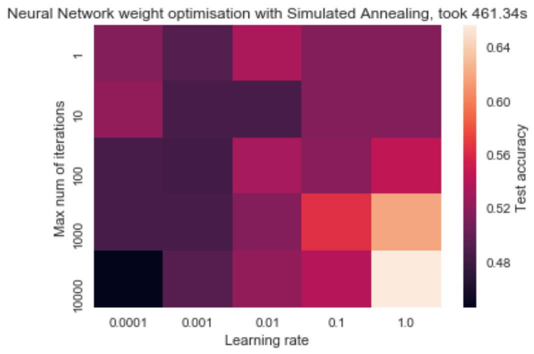
The time taken is the combined time elapsed for all of the 5×5 trials of optimisation. As we shall see the running time of later experiments we can compare in this dimension as well.

B. Simulated Annealing (SA)

Note that the `mlrose` library does not provide direct control to temperature, instead it manages the decrement in temperature but allows the choice of the manner of decay. We test on both *ExpDecay* an exponential decay schdeule, and *GeomDecay* a geometric decay schedule, to see if the choice would make performance difference.

Intuitively and as explained in the corresponding lecture, the exponential decay schedule appears preferrable: that in each iteration, the search point (i.e. the metal particle by analogy) becomes exponentially more difficult to climb up or down the optimisation landscape, and that the hills and valleys are augmented in the same exponential manner over time or temperature.





First heatmap: with ExpDecay, Second heatmap: with GeomDecay

Most performant settings:

- `schedule` = `ExpDecay`
- `learning_rate` = 1.0
- `max_iter` = 10000
- gives 68% test accuracy
- overall takes 479 seconds

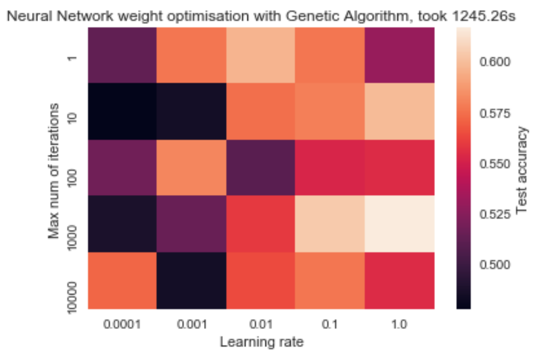
The same heatmaps as that of RHC is plotted for direct comparison. We can see basically a similar heat pattern as RHC. Different from RHC, its optimal weight is attained at `max_iter = 10000`, and that the best test accuracy (from *ExpDecay*) is slightly below RHC. This shows that both SA and RHC are resonably good optimiser to use, compared to the sklearn baseline.

Now compare between the two heatmaps, first notice that the test accuracy scale is different, that with the same brightest cell at the bottom right corner, *ExpDecay* gives a higher accuracy 68% $>$ 65%. Thus empirically speaking, the choice of *ExpDecay* is also justified. To provide an explantion, geometric decay works by shrinking the (height of) hills and valleys by a constant multiplcative rate, but this does not align as great as exponential decay does, with stabilising search points to the maxima.

SA takes twice as much time as RHC does, and the test accuracy did not surpass by much. Hence the simpler and faster algorithm, RHC, is more preferrable.

C. Genetic Algorithm (GA)

In exploring hyperparameters: Population size controls the degree of crossover, that drives the combination and improvement of previous generations of solutions; Mutation probability defines how probable will each element of the state vector flip. These will be searched on in three trial runs.

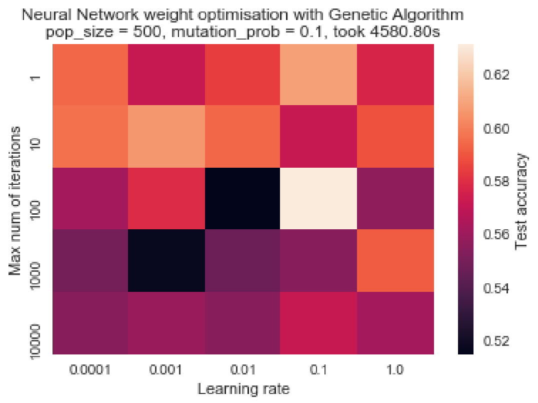


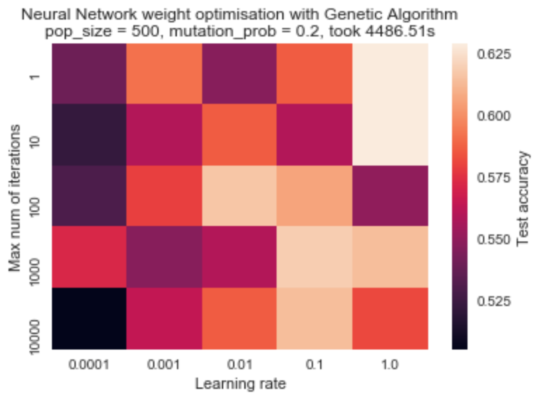
First heatmap: with pop_size = 200, mutation_prob = 0.1

Second heatmap: with pop_size = 500, mutation_prob = 0.2

Third heatmap: with pop_size = 500, mutation_prob = 0.2 Note: two erattas

```
- lr=1, max_iter=1:
  - Test accuracy:  0.504950495049505
```





```
- lr=1, max_iter=10:
  - Test accuracy:  0.5643564356435643
```

Most performant settings:

- `pop_size` = 500
- `mutation_prob` = 0.1
- `learning_rate` = 0.1
- `max_iter` = 100
- gives 63% test accuracy
- overall takes 4580 seconds

Hence we can see that the best setting amongst is the one with `pop_size = 500` and `mutation_prob = 0.1`, giving 63% test accuracy. This is because with a larger population at start, more combinations of solutions can be tested per iteration, giving a better weight. Note that for both the second and third heatmap with `pop_size = 500`, the test accuracy scale is shifted upwards from the first heatmap with `pop_size = 200`, indicating that increase in population size is useful. Mutation probability on the other hand does not seem to contribute much to further raising the test accuracy, but the setting is found to be different: `lr = 0.01`, `max_iter = 10000`. This speaks out that crossover has a greater effect over improving the population than mutation (flipping bits) without loss of generality.

Finally, the overall 25 trials took a ridiculously long time to run, yet it does not surpass RHC or SA in test accuracy, so GA is not deemed the optimal algorithm to use in this scenario.

D. Conclusion on weight optimisation

RHC is the best algorithm for this task, as reflected by relatively high test accuracy and quick running time. It works with the setting `learning_rate = 1.0` and `max_iter = 1000`. It is similar to gradient descent in the sense that unlike SA, it always picks the best move to optimise the weights, just as how gradient descent chooses the negative direction of the greatest gradient. This also corresponds to a smooth weight parameter landscape in which hill climbing with continuous moves is suited for.

II. THREE OPTIMISATION PROBLEMS

Setup

- All initial states are random

A. Problem 1: OneMax

Illustrates Genetic Algorithm (GA).

The OneMax problem seeks the vector $x = [x_1, x_2, ...x_{n-1}]$ that maximises the sum of all entries (fitness)

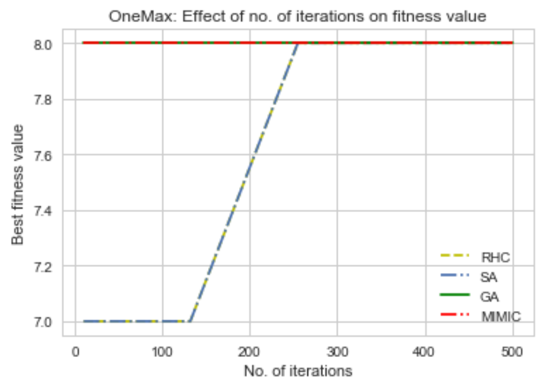
$$f(x) = \sum_{i=0}^{n-1} x_i$$

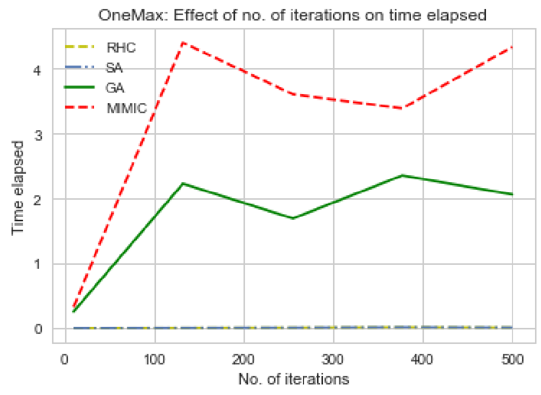
This is a common example of a toy but illustrative randomised optimisation problem, because there is one global (obvious to humans) optimum $x^* = [1, 1, ..., 1] = \mathbb{1}$ that is left to the algorithm to explore and search for. This problem has the advantage for quick prototyping for testing algorithms too.

For RHC and SA, they are to climb an $n = 8$ dimensional space (hypercube) to land at the destination.

For GA, this means having multiple partial solutions of 1's at various location and combining them by crossover. An edge of GA over RHC and SA is that in a sense RHA and SA are *forgetful*, in that they do not incorporate information from previous attempts to form a better solution the next time, but GA does so by breeding.

For MIMIC, the probability density (technically probability mass in this discrete case) is estimated successively so that more and more mass will accumulate on the x that can maxmimise $f(x)$, through the use of a generated dependency tree to sample the entire density $P(x)$ each iteration, building on previously known conditional dependences. Here the conditional dependence might be interpreted as spreading 1's in the x vector such that a vector with all 1's is most encouraged by the respective parents of tree nodes.





These two charts depict the performance of each four algorithms in two aspects of the same experiments:

- Best fitness value attained
- Time elapsed for the search

which are both parameterised on

- Maximum number of iterations

Since this is a maximisation problem, and the optimal fitness is simply $f^* = 8$, we can see how GA and MIMIC always hit the correct answer with any number of iteration (5 ticks along `[0, 500]` inclusive). SA and RHC experiences common fate, where at the first two smaller settings of maximum iterations, they settle on a suboptimum of $f(x) = 7$. As more iterations are permitted, they quickly flock to the expected $f(x) = 8$.

In terms of time, RHC and SA handles time complexity well since they do not scale up with number of iterations. MIMIC is expected to suffer from long running time since the density estimation task requires mimium spanning tree building and other expensive subroutines. GA is somewhat in between no-time estimations and the longer one. This is because the crossover operator would suffice in terms of complexity to handle this problem, to combine solutions quickly. MIMIC definitely overkills the task.

To elaborate more on crossover, note that two partial solutions for OneMax can be completely unrelated and far from one another in the searching space, and the crossover operator do not care much on such locality by still gluing the two solutions together. This achieves the effect of jumping across the search space instead of being locked down in the current neighbourhood.

As a result, GA is recognised to be the best algorithm for OneMax.

B. Problem 2: Travelling Salesman Problem (TSP)

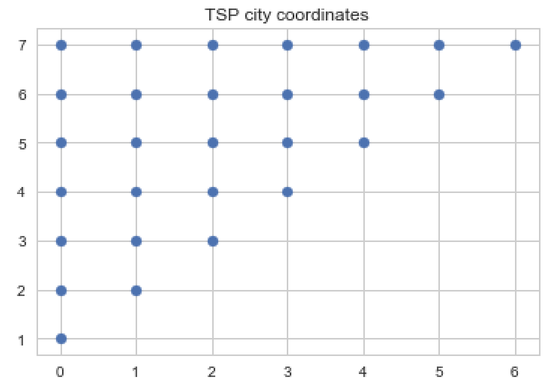
Illustrates Simulated Annealing (SA).

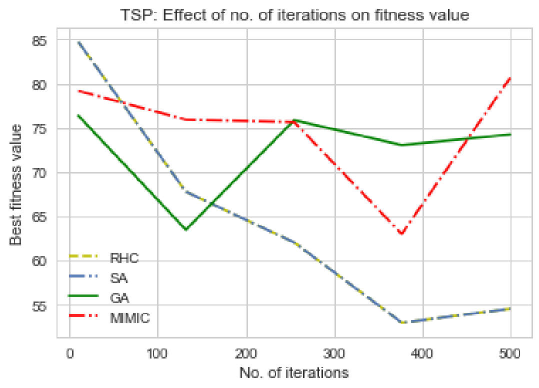
The travelling salesman problem is a classic puzzle and is famous for being NP-hard in general. We approach this therefore in a randomised/simulated way instead of using a closed form general solver.

This problem is to find a shortest path at which all cities (specified in input as nodes in a graph) are visited exactly once. This is reducible to finding a permutation of the input list of cities that minimises travel distances in between.

$$f(x) = \text{total distance travelled according to tour x}$$

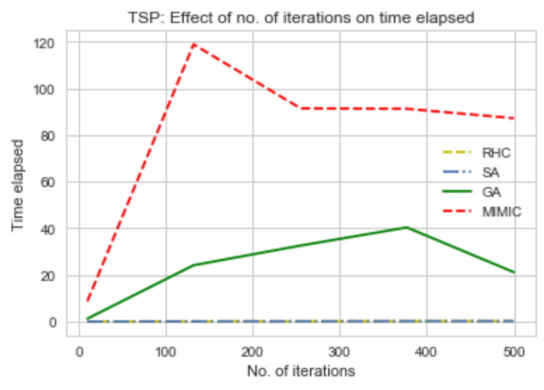
In this particular problem instance, the following cities are used as input. This assumes a complete graph, in which every city is reachable by every other city on the 2D plane, and that the coordinates define their relative locations, meaning that their distance is euclidean distance.





These charts follow the same format as Problem 1. The plots of best fitness value are interesting, in that severeal patterns can be observed:

- At a smaller number of iterations, GA outperforms all other algorithms, due to its ability to combine partial local solutions of shortest travelling paths of subsets of cities. RHC and RA are close to that of GA. MIMIC stays higher up in the fitness value, stuck at estimating density but the complication did not help, as seen as its high time complexity requirement too.
- At higher number of iterations, MIMIC finally reaches a fitness on-par with the previous



fitness by GA, but GA bounced up to suboptimals. Meanwhile, SA (and RHC) strives to reach an even lower fitness value that surpasses both of GA and MIMIC. This illustrates the power of trial-and-error in algorithms like hill climbing, that sometimes elaborate planning might not be of great use, and getting cheap explorations could be proven useful.

Considering both a minimum fitness value given enough iterations, and less time elapsed, SA is the best algorithm for solving TSP.

C. Problem 3: EightQueens

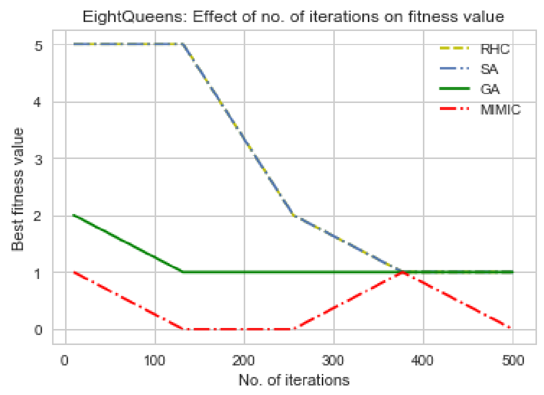
Illustrates MIMIC.

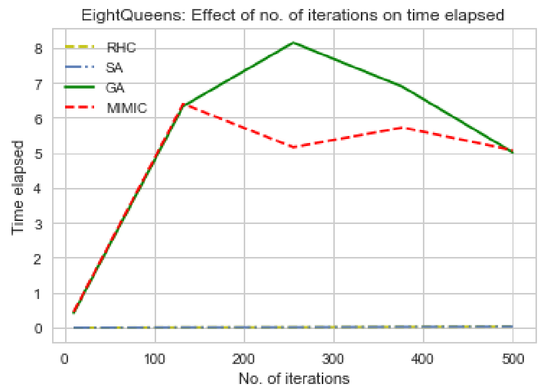
Queens here refer exactly to those in Chess, and they attack any piece that is either along the same vertical or horizontal line with it, or is along any of the two diagonal lines extending from the queen. It is a corollary that if one queen is attacking another queen, the other queen must also be attacking the first queen, so attacking queens at least come in pairs. Now on a 8-by-8 chess board there are only 8 queens, and there are several arrangements such that no queen would attack any other queen, this optimisation problem prompts to find one of those. In fact

$$f(x) = \text{no. of pairs of attacking queens on board x}$$

where $x = [x_0, x_1, ..., x_7]$ and x_i represents the row position (0, 1, ... 7) of the queen ith column queen, which in other words sits at coordinate (i, x_i) column-by-row. We are to minimise this cost.

Since this problem also has substructures that some subset of non-attacking queens can form the basis of the full solution, we are expecting better performance from GA and MIMIC.





From these charts, we can see that MIMIC is the only algorithm that can give the best chess arrangement of getting 0 attacking queens at 3 out of 5 times during the 5 different number of maximum iterations. Other algorithms could only give at best a solution with one pair of attacking queens remaining.

Notably RHC and SA leverages the iterations to attain a much better result than initially. GA flattens out at 2. Only MIMIC can precisely solve this problem that involves combination and somewhat a domino effect, since moving a queen to test on another position would definitely affect the unmoved queens, possibly their non-attacking status. EightQueens thus falls into the kind of problems that requires deliberate planning to solve in full. MIMIC prefectly demonstrates its strength through the use of density estimation and using conditional dependence information.

Regarding running time, surprisingly GA runs for longer than MIMIC in higher number of iterations, making MIMIC relatively speaking more efficient. RHC and SA continues to compute in constant time to number of iterations.

As a result, MIMIC is the best algorithm for solving EightQueens.

III. CONCLUSION

After the survey on various optimisation problem across the 4 randomised optimisation algorithm, we come to realise the respective strengths and weaknesses of the algorithms used in different situations:

- Neural Network weight optimisation: RHC
- Linear and combinatory problem (OneMax): GA
- Inherently exploratory problem (TSP): SA/RHC
- Inherently highly interdependent problem (EightQueens): MIMIC

Of course, there is no definite answer to a general-case algorithm, but the lesson is that employ MIMIC when the task needs careful precise moves to optimise and time is not a significant concern, otherwise depending on the problem's nature inclining towards combining subsolutions and exploration, pick GA and SA/RHC respectively. Since there is No Free Lunch, domain knowledge injection is made at the entrance of the fitness function which we have to consider case-by-case for optimality.