dhsieh6@gatech.edu

Randomized Optimization

Throughout this assignment, I used ML Rose, an existing package for Python that implements search algorithms and associated optimization problems.

Part I: Optimizing Neural Networks using Search Algorithms

The effectiveness of neural networks is driven by the weights chosen for model parameters, so developing methods to optimize weights and biases in neural networks directly impacts performance. A common way to optimize these weights and biases is to use gradient descent. The gradient descent, which is calculated as a partial derivative, represents the rate of change. It allows us to reach a local maximum, but the problem is that for complex equations (like neural networks), there is a high risk of reaching only a local, not global maximum. Search algorithms introduce interesting techniques to get around this, so in the upcoming section, I explore three search algorithms: randomized hill climbing, simulated annealing, and a genetic algorithm. I apply them to optimizing a neural network, and compare the results with that achieved by gradient descent.

I applied the search algorithms to the same neural network problem space as in the last assignment. Specifically, I have a dataset of reddit comments that were either removed or not removed by human moderators. The reddit comments themselves are converted to numeric vectors using tf-idf. The neural network attempts to predict whether or not comments that show up in the testing data set were removed or not.

First, I wanted to figure out how large of a sample to use here. I leveraged just the random hill climbing algorithm to do this. I used mostly default parameters for this, with zero restarts.

2000 samples				
Time to fit model			174.6 sec	
Time to predict model			0.1 sec	
Training accuracy			0.514	
Testing accuracy			0.587	
	Precision	Recall	F1-Score	Support
Not Removed	0.58	0.19	0.29	1221
Removed	0.38	0.79	0.51	779

3000 samples				
Time to fit model			331.0 sec	
Time to predict model			0.2 sec	
Training accuracy			0.500	
Testing accuracy			0.607	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.92	0.73	1820
Removed	0.38	0.08	0.13	1180

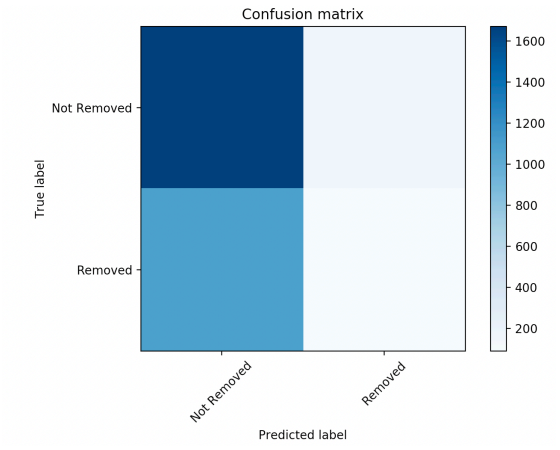
5000 samples				
Time to fit model			680.2 sec	
Time to predict model			0.51 sec	
Training accuracy			0.619	
Testing accuracy			0.607	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	1.00	0.76	3037
Removed	0	0	0	1963

The tables show the different results achieved with a neural network with 3 hidden layers optimized with the random hill climb algorithm for sample sizes [2000, 3000, 5000]. Although the neural network trains faster on 2000 samples, the results improve with 3000 samples. The problem is that the results improve with 3000 samples by essentially choosing “Not Removed” more often. This is evident in the 3000 samples table where Recall for Removed is just 0.08 compared to 0.79 with 2000 samples. This only gets worse at 5000 samples, where the model no longer predicts any comments are removed. Even balancing the dataset did not help. Based on these results, I decided that 3000 samples was a big enough dataset to use to compare the different search algorithms.

Random Hill Climb

As discussed earlier when trying to figure out how many samples to use for the rest of this analysis, I discovered that it appeared that random hill climb was converging to a local optimum. Essentially, it learns that by guessing that a comment isn't removed, it will do better than chance because there are more not removed comments than removed ones. In fact, it appears as if the hill climb, which in the above analysis did not feature any restarts, reached a local maximum and was not able to overcome it.

To address the issue, I increased the number of restarts allowed to 10, with the hopes of forcing the random hill climb to move out of the local optimum.



3000 samples				
Time to fit model			865.8 sec	
Time to predict model			0.4 sec	
Training accuracy			0.60	
Testing accuracy			0.59	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.92	0.73	1820
Removed	0.38	0.08	0.13	1180

Unfortunately, this did not improve results as expected, and instead just made the running time much longer. This is possibly because the random restarts did not actually move the algorithm out of a local maximum. The problem with the random hill climb is that although it explores the space more, it does so locally, and is thus still very much subject to the local maximum issue. To attempt to fix it, I increased the number of attempts the algorithm would try in order to find a different maximum, but this also did not help, since the algorithm already thinks that it has found a maximum. The results remained the same. Doing so also increased run-time, so it wasn't very effective overall. I tried reducing the learning rate to 0.001, and that also did not help. Same results! My overall conclusion is that the neural network weight optimization is too complex, and the random hill climb continues to get stuck at a local maximum.

Simulated Annealing

For the simulated annealing algorithm, I similarly tried to run it with the given defaults provided by ML Rose for geometric decay. The results are no different from the initial results achieved through random hill climb. As observed in the below table on the left, the model similarly optimizes for "Not Removed." I tried again with a different geometric decay function with a 0.999 decay. Still no difference! The reason for this is that at low temperatures, simulated annealing behaves very much like random hill climbing, and ultimately lands at a local optima. As you can see, the local optima it lands at is actually the same as the one found with the random hill climb.

Since simulated annealing can converge to local maximums if the temperature declines too quickly, I adjusted the decay to grow even slower at 0.99999.

3000 samples		**DEFAULT: 0.99 decay**		
Time to fit model			591.2 sec	
Time to predict model			0.29 sec	
Training accuracy			0.601	
Testing accuracy			0.587	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.92	0.73	1820
Removed	0.38	0.08	0.13	1180

3000 samples		**0.999 decay**		
Time to fit model			962.86s	
Time to predict model			0.59 sec	
Training accuracy			0.601	
Testing accuracy			0.587	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.92	0.73	1820
Removed	0.38	0.08	0.13	1180

3000 samples		**0.99999 decay**		
Time to fit model			476 sec	
Time to predict model			0.59 sec	
Training accuracy			0.514	
Testing accuracy			0.588	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.88	0.72	1820
Removed	0.43	0.14	0.21	1180

Genetic Algorithm
I did the same analysis for the genetic algorithm. I used a population size of 200, which was the default provided by ML Rose. The algorithm performed better than all the above search algorithms, which isn't unexpected since genetic algorithms do a better job of exploring random spaces. Since neural networks are so complex, it is not surprising that the genetic algorithm does better. However, the genetic algorithm is significantly slower than the other search algorithms, which is also a reason that it doesn't scale well for large and complex datasets.

3000 samples		**Population Size: 200**		
Time to fit model			6954.3 s	
Time to predict model			0.24 s	
Training accuracy			0.611	
Testing accuracy			0.60	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.95	0.74	1820
Removed	0.48	0.07	0.13	1180

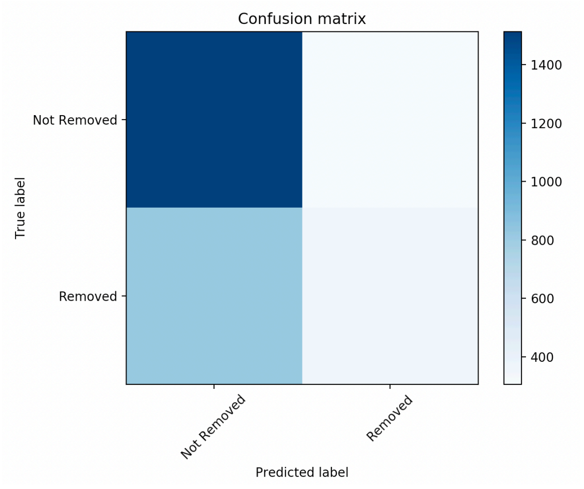
3000 samples		**Population Size: 100**		
Time to fit model			2899 s	
Time to predict model			0.24 s	
Training accuracy			0.50	
Testing accuracy			0.60	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.96	0.74	1820
Removed	0.40	0.04	0.07	1180

3000 samples		Population Size: 400		
Time to fit model			9298.96s	
Time to predict model			0.199s	
Training accuracy			0.50	
Testing accuracy			0.45	
	Precision	Recall	F1-Score	Support
Not Removed	0.61	0.28	0.38	1820
Removed	0.39	0.73	0.51	1180

I tried a couple different population sizes to understand the impact of that on overall model performance. Interestingly, it doesn't really improve testing accuracy, but it actually moves the results out of the local maximum of always choosing "Not Removed." This is likely because a larger population size also allows for more cross population during the evolution stages.

Gradient Descent
Finally, I wanted to compare the above optimizations with basic gradient descent to figure out if using a search algorithm actually results in better performance. The performance here was actually significantly better, both for training accuracy and testing accuracy. This is likely because gradient descent is able to shoot past local maximum if the learning rate is tuned properly.

3000 samples				
Time to fit model			611.56 sec	
Time to predict model			0.23 sec	
Training accuracy			0.789	
Testing accuracy			0.625	
	Precision	Recall	F1-Score	Support
Not Removed	0.65	0.83	0.73	1820
Removed	0.54	0.31	0.39	1180



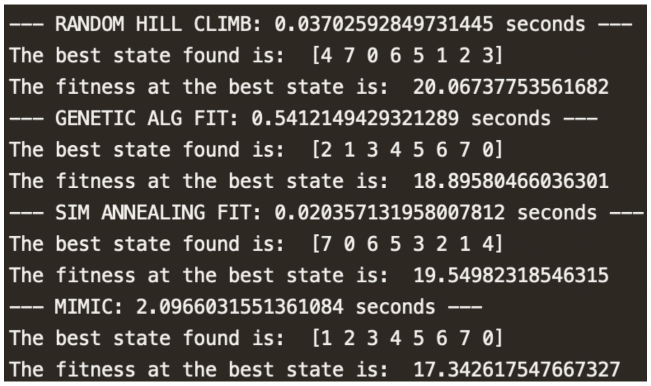
Part II: Defining problem domains for different search techniques

Each search technique excels at handling different optimization problems.
- Genetic Algorithm:
  - Excels at combinatorial optimization problems that can solve complex problems by combining three basic operators
  - Also enables more complex searching through random exploration via crossover
  - Not as efficient for larger datasets, and must match underlying problem set well
- Simulated Annealing:
  - Better at avoiding local optimums by using a "temperature" measure that determines whether or not to accept a given maximum
  - Can handle noisy and complex data sets
  - However, had to tune for the right temperature setting
- MIMIC

- Uses a density measurement to guess where exploration should occur
- Performs well for complex datasets that have multiple local maxima - mainly because it can interpret the underlying structure of the dataset

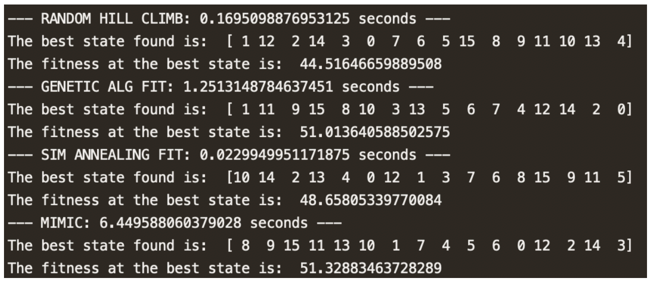
Traveling Salesperson Problem: Genetic Algorithm
The traveling salesperson problem tries to optimize the shortest tour a salesperson takes starting and ending in the same city, and visiting all other cities exactly once. The traveling salesperson problem is hard because it get exponentially more complex as an additional city is added. The amount of states to explore increases and the calculation of distance is a complex one.

The hypothesis is that an algorithm like GA would perform better in this situation because it is great at exploring larger spaces. The downside is that GA typically takes longer to calculate than other search algorithms. Similarly, MIMIC should also similarly perform well, but less well as the problem gets more complex, since the probability distribution space should be harder to represent mathematically. Meanwhile, random hill climbing and simulated annealing algorithms, which are more likely to get stuck at local optima, should perform worse. They will, however, likely perform faster.



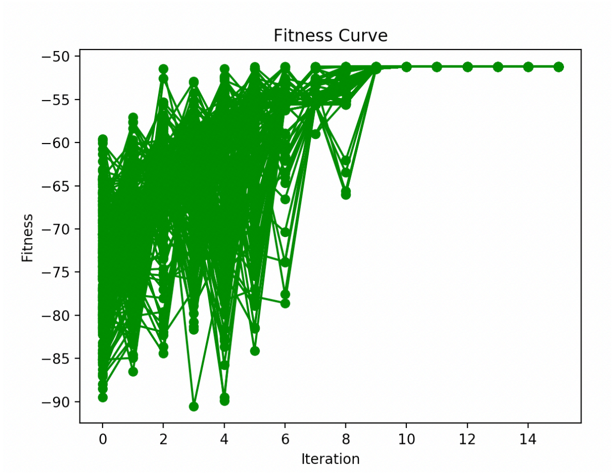
```
--- RANDOM HILL CLIMB: 0.03702592849731445 seconds ---
The best state found is:  [4 7 0 6 5 1 2 3]
The fitness at the best state is:  20.06737753561682
--- GENETIC ALG FIT: 0.5412149429321289 seconds ---
The best state found is:  [2 1 3 4 5 6 7 0]
The fitness at the best state is:  18.89580466036301
--- SIM ANNEALING FIT: 0.020357131958007812 seconds ---
The best state found is:  [7 0 6 5 3 2 1 4]
The fitness at the best state is:  19.54982318546315
--- MIMIC: 2.0966031551361084 seconds ---
The best state found is:  [1 2 3 4 5 6 7 0]
The fitness at the best state is:  17.342617547667327
```

The table on the left shows the different runtimes and results for each of the search algorithms. The problem space here was simple - only 8 cities to optimize. I used the following parameters for each model: RHC (max_iters=inf, restarts=0), GA (pop_size=100, mutation_prob=0.1, max_attempt=10, max_iters=inf), SA (decay=0.99), MIMIC (pop_size=200, keep_pct=0.2). As expected, genetic algorithms and MIMIC performed very similarly. They also took longer to calculate than the other two search algorithms. For these results, I included 2 restarts in the random hill climb to try to avoid local maxima. I also lowered the population size for genetic algorithms to 100, since our problem space is smaller. Otherwise, all other algorithms used typical default values. Based on the results above, I guessed that the genetic algorithm was actually choosing the wrong optimum, since it suffers from being unable to figure out when it is “right”. So, I adjusted the mutation level from 0.1 to 0.3 to see if that would change the results. Doing so actually resulted in the same fitness maximization value, but a completely different path when compared to MIMIC. Interestingly, by increasing the maximum iterations to 20 for genetic algorithms and random hill climb, I was able to bring both to around 17-18 in terms of fitness, depending on the randomness of each run. This again demonstrates how both have a chance of either optimizing towards a local maximum, or stopping at the wrong iteration.



```
--- RANDOM HILL CLIMB: 0.1695098876953125 seconds ---
The best state found is:  [ 1 12  2 14  3  0  7  6  5 15  8  9 11 10 13  4]
The fitness at the best state is:  44.51646659889508
--- GENETIC ALG FIT: 1.2513148784637451 seconds ---
The best state found is:  [ 1 11  9 15  8 10  3 13  5  6  7  4 12 14  2  0]
The fitness at the best state is:  51.013640588502575
--- SIM ANNEALING FIT: 0.0229949951171875 seconds ---
The best state found is:  [10 14  2 13  4  0 12  1  3  7  6  8 15  9 11  5]
The fitness at the best state is:  48.65805339770084
--- MIMIC: 6.449588060379028 seconds ---
The best state found is:  [ 8  9 15 11 13 10  1  7  4  5  6  0 12  2 14  3]
The fitness at the best state is:  51.32883463728289
```

To test my earlier hypothesis that GA would perform better for complex spaces, I defined more cities, creating a total list of 16 cities. With the larger dataset, GA does indeed perform slightly better than MIMIC. It is also significantly faster than MIMIC in terms of processing time. Surprisingly though, with this more complex dataset, random hill climb and sim annealing fit



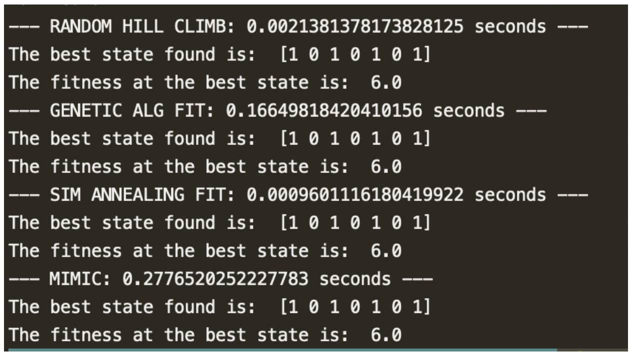
actually perform better. This is possibly because the genetic algorithm and MIMIC terminate too quickly.

However, when observing the fitness curve for MIMIC, this does not appear to be true.
In fact, when observing the different fitness curves generated by each algorithm, MIMIC is quite efficient in converging - needing only around 10 iterations whereas other models needed around 30. However, each iteration is computationally more expensive, so ultimately, it is still a more inefficient algorithm.

Conceptually, it seems unlikely that the hill climbing algorithms (random and sim annealing) perform better, so it is likely that the parameters I set are effecting my ability to compare the different algorithms.

Flip Flop Problem: Simulated Annealing

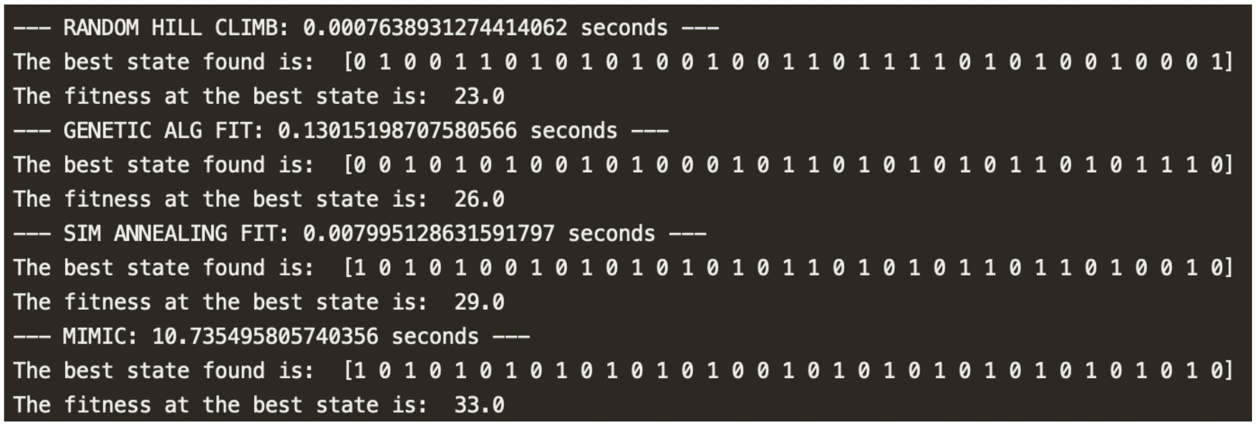
The Flip Flop problem calculates the number of alternating bits. The problem is difficult because it has a large number of local maximums, which makes it hard for search algorithms. I used the following parameters for each model: RHC (max_iters=inf, restarts=0), GA (pop_size=100, mutation_prob=0.1, max_attempt=10, max_iters=inf), SA (decay=0.99), MIMIC (pop_size=200, keep_pct=0.2).



```
--- RANDOM HILL CLIMB: 0.0021381378173828125 seconds ---
The best state found is:  [1 0 1 0 1 0 1]
The fitness at the best state is:  6.0
--- GENETIC ALG FIT: 0.16649818420410156 seconds ---
The best state found is:  [1 0 1 0 1 0 1]
The fitness at the best state is:  6.0
--- SIM ANNEALING FIT: 0.0009601116180419922 seconds ---
The best state found is:  [1 0 1 0 1 0 1]
The fitness at the best state is:  6.0
--- MIMIC: 0.2776520252227783 seconds ---
The best state found is:  [1 0 1 0 1 0 1]
The fitness at the best state is:  6.0
```

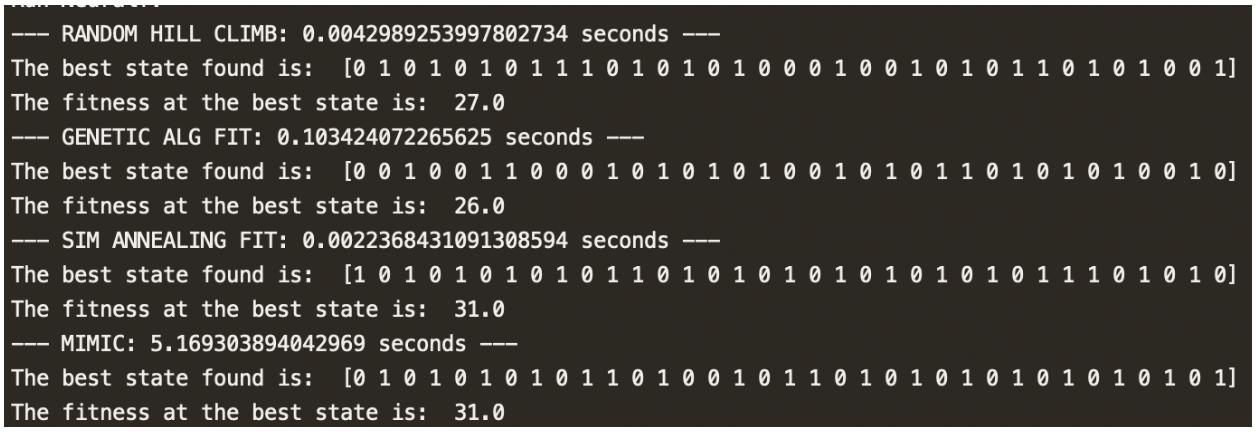
For the simple space with only 7 elements, all of the search algorithms are able to maximize result, with sim annealing feat doing it the fastest in terms of computational time.

In order to push the algorithms furthered, I used a more complex data set of 35 elements. The results are shown in the table below.



```
--- RANDOM HILL CLIMB: 0.0007638931274414062 seconds ---
The best state found is:  [0 1 0 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 0 1 0 0 1 0 0 0 1]
The fitness at the best state is:  23.0
--- GENETIC ALG FIT: 0.13015198707580566 seconds ---
The best state found is:  [0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 1 0 1 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 1 0]
The fitness at the best state is:  26.0
--- SIM ANNEALING FIT: 0.007995128631591797 seconds ---
The best state found is:  [1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 1 0 1 0 0 1 0]
The fitness at the best state is:  29.0
--- MIMIC: 10.735495805740356 seconds ---
The best state found is:  [1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0]
The fitness at the best state is:  33.0
```

As expected, performance here is actually quite different. In fact, the random hill climb performs the worst, likely because there are a lot of local maximums in the flip flop problem. The random hill climb is unable to identify these local maximums. MIMIC does well, since it can understand the underlying pattern of the dataset. However, for all of these algorithms, the hypothesis is that performance could improve if we continue to iterate for longer periods of time and increase the randomness in the calculations. To adjust this, I went with the following parameters: RHC (max_iters=inf, restarts=5), GA (pop_size=100, mutation_prob=0.3, max_attempt=10, max_iters=inf), SA (decay=0.99), MIMIC (pop_size=200, keep_pct=0.2).

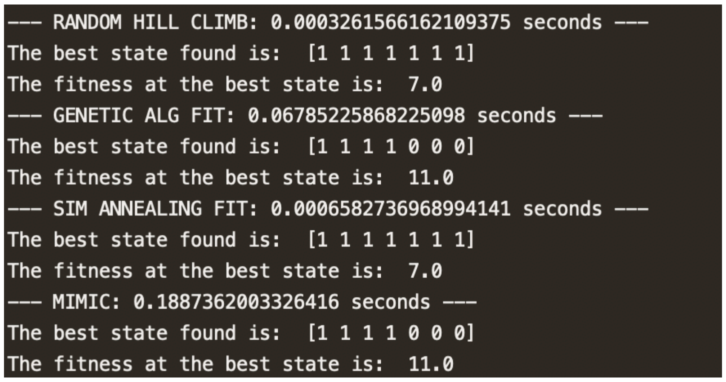


```
--- RANDOM HILL CLIMB: 0.0042989253997802734 seconds ---
The best state found is:  [0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 0 1]
The fitness at the best state is:  27.0
--- GENETIC ALG FIT: 0.103424072265625 seconds ---
The best state found is:  [0 0 1 0 0 1 1 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0]
The fitness at the best state is:  26.0
--- SIM ANNEALING FIT: 0.0022368431091308594 seconds ---
The best state found is:  [1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0]
The fitness at the best state is:  31.0
--- MIMIC: 5.169303894042969 seconds ---
The best state found is:  [0 1 0 1 0 1 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
The fitness at the best state is:  31.0
```

Making these adjustments improved performance for the random hill climb, since the added randomness helps the algorithm avoid local maximums. For genetic alg fit, the Andrea in mutation probability seems to have caused too much randomness. Both simulation annealing fit and MIMIC now get the same results. However, this appears to be due to some randomness. In a later iteration of this project, I would run multiple trials to see how randomness impact measurements. Given that both simulation annealing and MIMIC find solutions with the same fitness level, and simulation annealing is significantly faster than MIMIC, simulation annealing appears to be the best algorithm for the flip flop problem.

Four Peaks: MIMIC

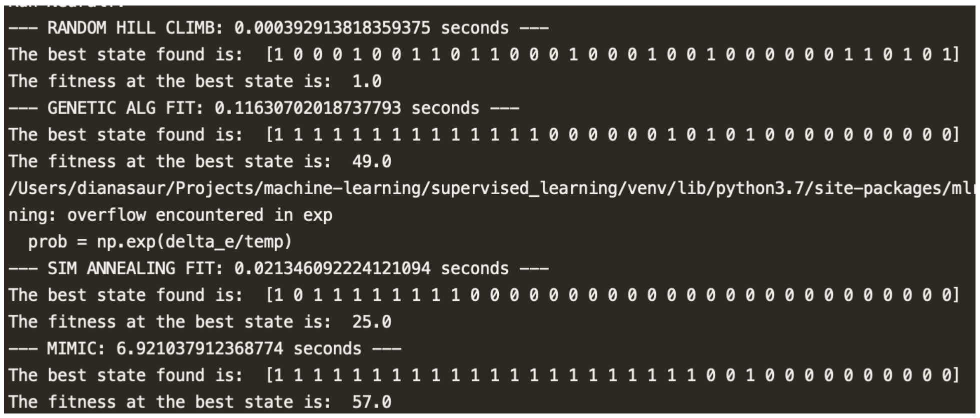
Finally, I applied the four different search algorithms to the four peaks optimization problem, which essentially has four local maximum. Because of the shape of the data distribution, my hypothesis is that MIMIC will be very good at optimizing for this problem. Random hill climb and simulation annealing will struggle with all the local maximum, whereas genetic algorithms are ultimately still optimizing for local maximums and may choose the wrong one for complex datasets. I used the following parameters for each model: RHC (max_iters=inf, restarts=0), GA (pop_size=100, mutation_prob=0.1, max_attempt=10, max_iters=inf), SA (decay=0.99), MIMIC (pop_size=200, keep_pct=0.2).



```
--- RANDOM HILL CLIMB: 0.0003261566162109375 seconds ---
The best state found is:  [1 1 1 1 1 1 1]
The fitness at the best state is:  7.0
--- GENETIC ALG FIT: 0.06785225868225098 seconds ---
The best state found is:  [1 1 1 1 0 0 0]
The fitness at the best state is:  11.0
--- SIM ANNEALING FIT: 0.000658273696899414 seconds ---
The best state found is:  [1 1 1 1 1 1 1]
The fitness at the best state is:  7.0
--- MIMIC: 0.1887362003326416 seconds ---
The best state found is:  [1 1 1 1 0 0 0]
The fitness at the best state is:  11.0
```

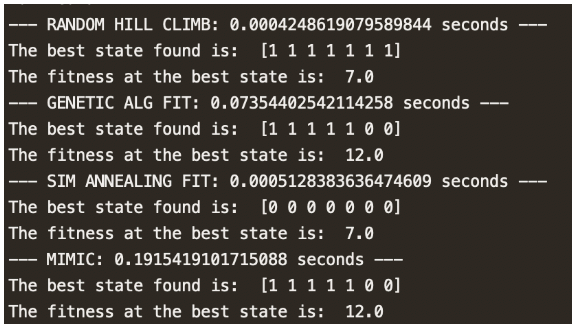
As expected, in this initial run, random hill climb and simulation annealing perform worse than the other two algorithms, which are better at avoid local maximum. Although they both run faster than the other algorithms, their performance gap does not merit using this algorithms.

To try to analyze this even further, I used the same parameter, but a more complex dataset. The expectation is that MIMIC will be able to understand the underlying distribution better, and pick a better maximum.

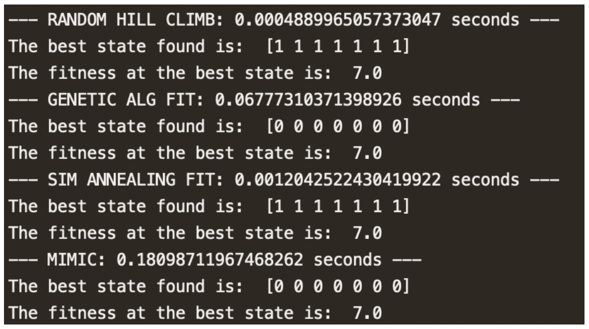


```
--- RANDOM HILL CLIMB: 0.000392913818359375 seconds ---
The best state found is:  [1 0 0 0 1 0 0 1 1 0 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1 1 0 1 0 1]
The fitness at the best state is:  1.0
--- GENETIC ALG FIT: 0.11630702018737793 seconds ---
The best state found is:  [1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0]
The fitness at the best state is:  49.0
/Users/dianasaur/Projects/machine-learning/supervised_learning/venv/lib/python3.7/site-packages/ml
ning: overflow encountered in exp
  prob = np.exp(delta_e/temp)
--- SIM ANNEALING FIT: 0.021346092224121094 seconds ---
The best state found is:  [1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
The fitness at the best state is:  25.0
--- MIMIC: 6.921037912368774 seconds ---
The best state found is:  [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0]
The fitness at the best state is:  57.0
```

As anticipated, MIMIC performs quite well here, clearly optimizing for better performance. Although it takes longer, the performance improvement makes it worth it. Unfortunately for genetic algorithms, which do not perform as well for complex datasets, the actual computation of it actually resulted in an overflow error.



```
--- RANDOM HILL CLIMB: 0.0004248619079589844 seconds ---
The best state found is:  [1 1 1 1 1 1 1]
The fitness at the best state is:  7.0
--- GENETIC ALG FIT: 0.07354402542114258 seconds ---
The best state found is:  [1 1 1 1 1 0 0]
The fitness at the best state is:  12.0
--- SIM ANNEALING FIT: 0.0005128383636474609 seconds ---
The best state found is:  [0 0 0 0 0 0 0]
The fitness at the best state is:  7.0
--- MIMIC: 0.1915419101715088 seconds ---
The best state found is:  [1 1 1 1 1 0 0]
The fitness at the best state is:  12.0
```



```
--- RANDOM HILL CLIMB: 0.0004889965057373047 seconds ---
The best state found is:  [1 1 1 1 1 1 1]
The fitness at the best state is:  7.0
--- GENETIC ALG FIT: 0.06777310371398926 seconds ---
The best state found is:  [0 0 0 0 0 0 0]
The fitness at the best state is:  7.0
--- SIM ANNEALING FIT: 0.0012042522430419922 seconds ---
The best state found is:  [1 1 1 1 1 1 1]
The fitness at the best state is:  7.0
--- MIMIC: 0.18098711967468262 seconds ---
The best state found is:  [0 0 0 0 0 0 0]
The fitness at the best state is:  7.0
```

As a final experiment, I wanted to test how well the four search algorithms performed for the four peaks problem with different threshold percentages. My expectation was that for smaller threshold percentages, genetic algorithms and MIMIC would perform better since they would be able to detect those difference with more randomized searches and through understanding the distribution of the underlying data structure. The left table above shows a threshold percentage of 0.01. The right one shows one of 0.8, where all of the algorithms get confused and choose a local optimum.