Project 2: Randomized Optimization

CS-7641 Machine Learning

Spring 2015

Name: Fujia Wu

GT Account: fwu35

1. Implementation of Optimization Tools

I use the ABAGAIL tool for this assignment. https://github.com/pushkar/ABAGAIL ABAGAIL includes the four local random search algorithms: *randomized hill climbing*, *simulated annealing*, *generic algorithm* and *MIMIC*. It is also possible to adjust the optimization parameters for these algorithms. To study different optimization problems, it is needed to write different fitness function (Evaluation Function in src/opt/example folder), and give the dimension and range of the input bit strings.

Table 1 lists the parameters that need to be specified in order for each algorithm to function. The fitness function should be given for any algorithm and should be different for each optimization problem. All other parameters are provided for the optimization algorithms to work better or worse. In the following, tuning these parameters will be performed and the results will be analyzed and discussed.

Table 1 Parameters needed for all four optimization algorithms

	Randomized Hill Climbing	Simulated Annealing	Generic Algorithm	MIMIC
Fitness Function	*	*	*	*
Neighbor Function	*	*		
Distribution Function	*	*	*	*
Temperature, Cooling		*		
Crossover Function			*	
Mutation Function			*	
Dependency Tree				*

2. Learning Weights from Neural Network

2.1 *Neural Network Optimization Method*

For a neural network, regardless what algorithms are used for finding the weights, the goal is the find a set of weights that fit the training data the best. Therefore, the supervised learning problem can be converted into an optimization problem. The fitness function, in this case, should be a measure of how close the outputs for a given set of training data measure the labels of the training data. Based on what we learned in supervised learning (Bayesian learning), for training data with normal error distribution the maximum-likelihood estimation is to maximize the inverse of the sum of square error. Therefore, a suitable fitness function (with a single multi-inputs single-output perceptron as an example) can be,

$$f(\omega_1, \omega_2, \ldots, \omega_N) = \frac{1}{\sum_k \left(\sum_i \omega_i x_{i,k} - y_k \right)^2} \qquad (1)$$

where ω_i are the weights of the neural network, $x_{i,k}$ are the ith feature of the kth sample, and y_k is the label of the kth sample.

Notice that weights in a neural network are continuous and real-valued instead of discrete, therefore the neighbor function, distribution function, crossover function and mutation function need to be designed to deal with continuous hypothesis space. To do this, I selected the distribution function (for initialization or random restart) for each weight ω_i to be a uniform distribution from -0.5 to 0.5, *i.e.*,

$$\omega_i^0 = \varepsilon(-0.5, 0.5) \qquad (2)$$

where ω_i^0 is the initial or restart ith weight and $\varepsilon(a, b)$ is a random variable uniformly distributed from a to b.

For the neighbor function (for randomized hill climbing and simulated annealing), I selected it to be a change in the weight by a small random amount,

$$\omega_i^{n+1} = \omega_i^n + \varepsilon(-0.5, 0.5) \qquad (3)$$

where ω_i^n is the weight for the nth fitness function evaluation. For each neighbor, only 1 weight value is altered, which is again randomly uniformly selected from all weights. This random means to decide a neighbor ensures that the possible instances can cover the entire continuous hypothesis space.

For crossover function (for generic algorithm), there is no difference between discrete hypothesis space and continuous hypothesis space. I selected to use a uniform crossover function, which uses a uniform random Boolean function to decide the bit from which parent to keep in the children.

Finally, for mutation function (for generic algorithm), I selected to be a change in one of the weights by a small random amount, similar to the neighbor function, *i.e.*,

$$\omega_i^n \leftarrow \omega_i^n + \varepsilon(-0.5, 0.5) \quad (4)$$

2.2 *Data Set and Network Structure*

The data I use for neural network learning is Car Evaluation Data Set from the UCI Machine Learning Repository. This is one of the two data sets I selected for Assignment 1. The data is a multi-class classification problem. There are 6 attributes: buying price, maintenance price, number of doors, capacity in terms of persons, size of luggage boot and safety estimate of the car. Each attributes contains discrete category values, such as high, med and low. The label values are also discrete category data, including four classes: unacceptable, acceptable, good and very good. There are 1728 instances, in total.

I first parse the string values in the original data into numerical values normalized from 0 to 1 for the features. For the four label values, I convert them into 4-dimensional Boolean vectors to accommodate ABAGAIL. For example, the line

vhigh,vhigh,2,2,small,low,unacc

is parsed into

[1.0, 1.0, 0.0, 0.0, 0.5, 0.0] : [0, 0, 0, 1]

In assignment 1 (implemented with Matlab), I used a network with 6 input layers (for 6 features), 10 hidden layers and 4 output layers for the 4 types of classification.

2.3 *Results and Analysis*

The above mentioned neural network learning methods were implemented in ABAGAIL via the following files:

NeuralNetworkOptimizationProblem.java
NeuralNetworkEvaluationFunction.java
RandomizedHillClimbing.java
SimulatedAnnealing.java
StandardGeneticAlgorithm.java
BatchBackPropagationTrainer.java
BackPropagationNetwork.java

I ran the learn procedure with a jython wrapper file neuralnetwork.py, with the following running command:

jython -J-cp [path-to-ABAGAIL.jar] neuralnetwork.py < car.data

For the initial run, I used 1000 training iterations for RHC and SA. The initial temperature and cooling exponent for SA is initially set to be 1E11 and 0.95. The population, cross over size and mutation size for GA is set to be 200, 100 and 10, respectively. The training iteration for GA is set to be 100 to have approximately the same fitness function

evaluation times as RHC and SA. The results of RHC, SA and GA are then compared with the back propagation (BP) algorithms.

Figure 1 shows the training and test error for all cases as a function of the training size. First of all, similar to what I have found in Assignment 1, neural network learning does not tend to over fit for this car evaluation data size. This is probably due to the fact that the data set contains little random noise. Therefore, it is seen that for all algorithms the testing and training error rates are quite close. Second, it is seen that among the 3 optimization algorithms, for approximately the same number of fitness function evaluations, RHC has the lowest error rates. SA and GA have approximately the same error rates, which are higher than RHC. But the error rate for RHC is still much higher than the back propagation learning algorithms. This is likely due to the fact that the number of training iterations (1000) is still too small.

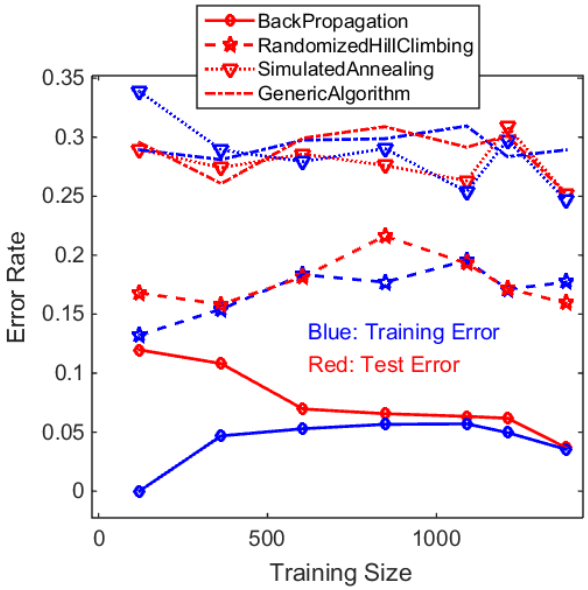


Figure 1: Comparison of the training and test error rate as a function of training size for BP, RHC (1000 training iterations), SA (1000 training iterations, 1E11 initial temperature, 0.95 cooling exponent), GA (100 training iterations, 200 populations, 100 cross over size, 10 mutation size)

To study how the optimization algorithms work as the number of training iterations increases, Figure 2 plots the training and test error rates for RHC and SA as the number of training iterations increases. It is seen that as more training iterations are performed, the error rates indeed decrease and approach the performance of the back propagation algorithm. For 100000 optimization iterations, the training error rates for RHC and SA are approximately 0.05, which is about the same as back propagation algorithm.

On the other hand, for optimization algorithms to achieve the same performance as back propagation, I noticed that considerably more time is needed. This is because the optimization algorithms (RHC, SA and GA) are all blind about the fitness function, *i.e.*, they

do not know how the fitness function look like. However, the back propagation algorithm is exactly based on the sum of square fitness function (the gradient decent rule). Therefore, back propagation algorithm can find the best weights faster than RHC, SA, and GA.

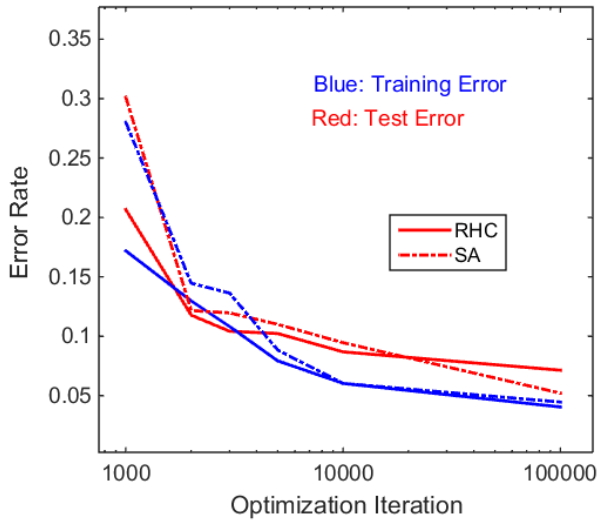


Figure 2: Comparison of the training and test error rate as a function of number of optimization iterations for RHC and SA (1E11 initial temperature, 0.95 cooling exponent)

3. Study of Three Optimization Problems

3.1 *k-Hills Problem*

The first optimization problem I create is called *k*-hills problem. As the name suggests, the fitness function is the sum of *k* Gaussian functions in an *n*-dimension input space, which gives the *k*-1 local peaks and one global peaks,

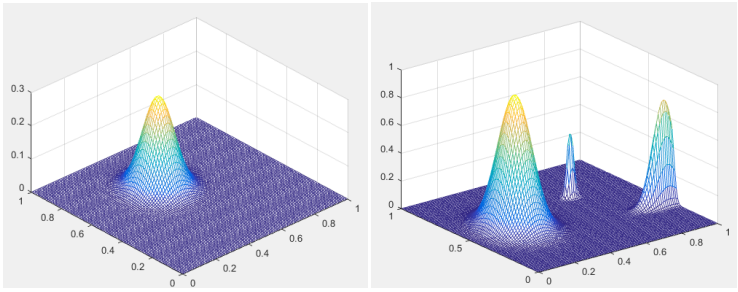
$$f(x_1, x_2, \ldots, x_N) = \sum_k h_k \exp\left[-\sum_n \frac{\left(x_n - \mu_{n,k}\right)^2}{2\sigma_{n,k}^2} \right] \qquad (5)$$

where h_k are the hill peaks, are $\mu_{h,k}$ and $\sigma_{h,k}$ the means and standard deviations of the *n*-dimension of the *k*th hills. In this problem, once *k* and *n* are given, parameters h_k, $\mu_{h,k}$ and $\sigma_{h,k}$ are randomly selected. Some examples of the function in two-dimensional input space are shown in Figure 3. It is seen that the optimization is inherently a hill climbing problem. Therefore, I am hoping SA and RHC will performance better than GA and MIMIC. However, as the number of hills increase, the problem becomes rather chaotic.

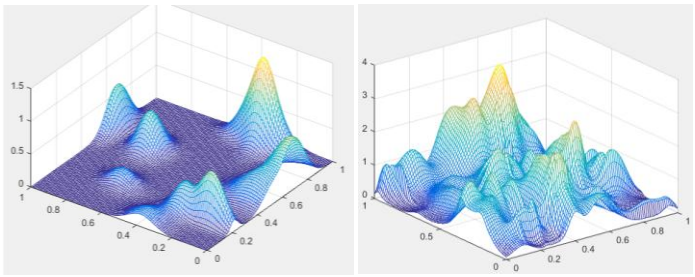
To discretize the problem, each input parameter are confined to take only integers between 0 (inclusive) and 100 (exclusive). First, let explore a simple case with $k = 1$ (1-hill problem). I implemented this fitness function as the java class KHillsEvaluationFuncion.java

and put it into the folder src/opt/example in ABAGAIL. I ran the learning procedure with a jython wrapper file khills.py, with the following running command:

```
jython -J-cp [path-to-ABAGAIL.jar] khills.java
```



(a) $k = 1$ (b) $k = 3$



(c) $k = 10$ (d) $k = 100$

Figure 3: Plots of two-dimensional function of k-hills function with $k = 1$, 3, 10, and 100

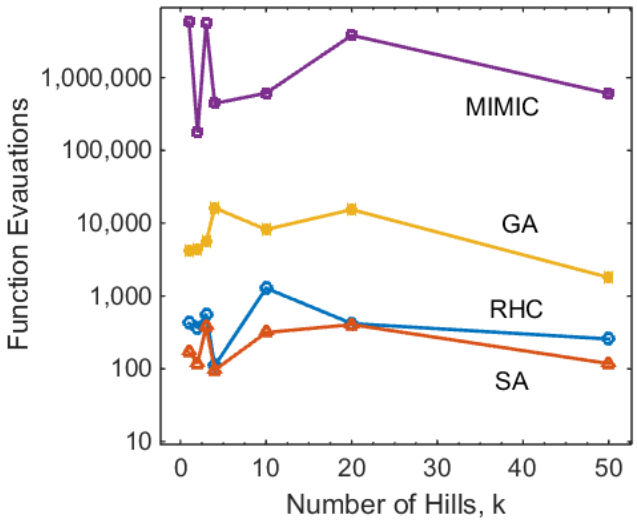


Figure 4: Required number of function evaluation needed to find the optimum for different optimization algorithms on the k-hills problem with $n = 2$ and different values of k.

Figure 4 plots the number of function evaluation needed to find the optimum for different optimization algorithms on the *k*-hills problem with $n = 2$ and different values of *k*. It is seen that for this problem, SA indeed performs the best, followed by RHC, GA and MIMIC. MIMIC, for this particular problem, needs a large number of function evaluations to find the optimum.

3.2 *Knapsack Problem*

The second optimization problem I selected is the Knapsack problem, which is already included in ABAGAIL. The problem is to find a set of items from a pool so that the total weight is maximized while keep the total value lower than a given value. The goal of selecting this problem is to highlight the performance of GA. Different from the *k*-hills problem, the Knapsack problem is not hill climbing in nature. Therefore, I do not expect RHC and SA will perform better than GA and MIMIC for this problem. In addition, this problem does not seem to have obvious "structure". Therefore, I expect MIMIC should not necessarily perform better than GA.

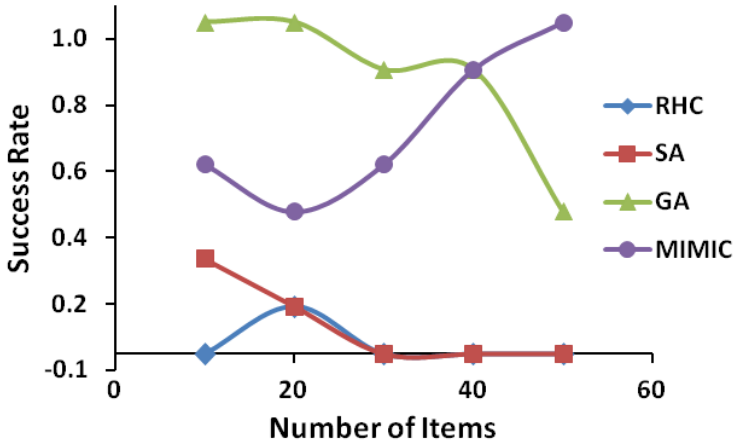


Figure 5: Success rate for finding the global optimum before the times of fitness function evaluation reaches $T = 1{,}000{,}000$ for the Knapsack problem. Detailed statistics are in the file knapsack_statistics.xlsx

It is difficult to make a comparison of different algorithms by just running each case once due to a number of random processes involved in this problem. First, for given number *N* items, the weight and volume of each item are randomly given. Second, the initial inputs, restart inputs (for RHC and SA) and initial population, mutation function (for GA and MIMIC) are also randomly selected. Therefore, for each run different algorithms perform differently. Therefore, to make a systematic study for any given problem I run each algorithm multiple times and calculate the averaged performance. For each Knapsack problem with a given *N*, I first find the global optimum by running each algorithm for extended amount of iterations. Then I defined the maximum of the optima from all algorithms to be the global optimum. I then run each algorithm multiple times for a fixed number of fitness function evaluations.

Then I calculate the averaged success rate that the algorithms can find this global optimum before the number of fitness function evaluations reaches a fixed value.

Figure 5 plots success rates for finding the global optimum using different algorithms before the times of fitness function evaluation reaches T = 1,000,000. Please refer to the file knapsack_statistics.xlsx for detailed measurement results and statistics. It is seen that clearly GA and MIMIC algorithms perform much better than RHC and SA. At least this indicates that more sophisticated methods do have advantages over simple hill climbing methods for certain problems! Comparing GA and MIMIC, it is seen from Figure 5 that for small number of items (N < 40), GA performs better than MIMIC. But for large number of items (N > 40), MIMIC has better performance than GA.

The reason for the different performance for GA and MIMIC at for different problem sizes, I think, is due to the fact that MIMIC assumes the features can be described by a dependency tree, *i.e.*, the value of one feature only depends on its parent and is independent of other features. When the number of items is small, there is really no freedom to choose the best items (weight and volume) and there is strong dependence between all items, *i.e.*, selection of one item changes the results of many other items. Therefore, in this case the dependence tree assumption is violated. This is probably why MIMIC does not perform as well as GA. But as the item size increases, there is more freedom for selecting the best items and there is less dependence among the features. In this case, MIMIC can gain information from its estimate of dependence tree and performs better than GA.

3.3 *k-Color Problem*

The third optimization problem I selected is the *k*-color problem for a random graph. This problem was discussed in the paper titled "*Randomized Local Search as Successive Estimation of Probability Densities*" by Prof. Charles L. Isbell. The problem is to find a coloring of for all the nodes of a graph that minimizes the number of adjacent pairs colored the same. The parameters of the problems are the number of colors, K, and a graph with number of nodes N and number of edges E. The input values are the color for each node. Therefore, there are K^N hypothesis in the input space.

The goal for selecting this problem is to highlight the advantage of MIMIC. This is because MIMIC is good at problems with structures. This *k*-color problem is indeed based on the clear structure relationships between different nodes.

I considered a directed graph with N nodes, each of which has M neighbors which are randomly for all other N-1 nodes. Therefore, they are in total $E = N*M$ directed edges. There are K available colors. The fitness function is defined as the inverse of the number of edges whose two connecting nodes have the same color. I implemented this fitness function in the KColorEvaluationFunction.java in the folder src/opt/example folder in ABAGAIL. I ran the

learning procedure with a jython wrapper file kcolors.py, with the following running command:

```
jython -J-cp [path-to-ABAGAIL.jar] kcolors.java
```

I considered a problem with 20 nodes (N = 20). I then vary the number of neighbors per nodes to investigate the performance of different optimization algorithms. The number of colors k is selected to be 5. Same with the previous knapsack problem, there are multiple random factors in each run, a meaningful metrics are the averaged ones among multiple experimental runs.

Figure 6 plots the success rate for finding the global optimum before the times of fitness function evaluation reaches T = 1,000,000 for different number of neighbors per node. The Detailed statistics are in the file kcolors_statistics.xlsx. Several observations can be made from Figure 6. First, different from what I expected, MIMIC does not perform better than the other 3 optimization methods. Second, SA and GA perform better than RHC and MIMIC. When number of neighbors is less than 10, SA performs the best. When the number of neighbors is greater than 10, GA performs the best. Thirds, the problem is relatively easy to optimize for ether very small or large number of neighbors, but difficult for values in between.

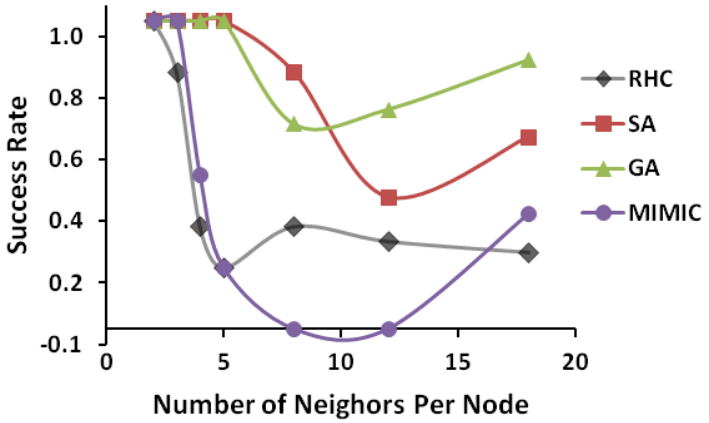


Figure 6: Success rate for finding the global optimum before the times of fitness function evaluation reaches T = 1,000,000 for k-color problem with N = 20 nodes for different number of neighbors per node. The number of colors k = 5. Detailed statistics are in the file kcolors_statistics.xlsx

It is surprising to find that MIMIC performs much worse than GA and SA for this problem. I originally thought the problem has well structure, and therefore MIMIC should be able to learn very well. But I guess this is one of the practical issues in Machine Learning in general, which is theory does not always work in practice! The real reason why MIMIC does not perform well on this problem is unknown to me. One possible reason I can think of is that the problem again may not satisfy the dependency tree rule. This is especially the case when

the neighbors are randomly determined for all the nodes. This causes each feature to depend on not one parent feature but many features, which make MIMIC hard to learn.

4. Summary

In this assignment, I used 4 optimization algorithms in ABAGAIL to study optimization problems: random hill climbing, simulated annealing, generic algorithms and MIMIC.

In the first task, I used the first 3 optimization algorithms to learn the weights for a neural network on a supervised learning problem I did in Assignment 1. To do that, a sum-of-square fitness function is used, and a randomized jumping method is used in the neighbor and mutation functions in the NN weights input space, which is continuous. I discovered that all 3 optimization algorithms can be used to learning the weights of NN and produces about the same training and test errors as the back propagation algorithms. However, using optimization methods take longer time then back propagation.

In the second task, I created and selected 3 optimization problems. The first one is a *k*-hills problem, which is simply sum of multiple multidimensional Gaussian functions. The problem is hill climbing in nature, and as expected, SA and RHC perform best. The second problem is the Knapsack problem. This problem does not have the hill climbing property, and as a result GA and MIMIC perform the best, at small and large problem size respectively. The third problem is the *k*-color problem on a graph with random neighbors. I designed this problem to highlight the performance of MIMIC (because it has structure); however, MIMIC does not perform as expected. This, again, highlights the general issue in Machine Learning: theory vs practice.

All the files need to reproduce the results in this assignment.

1. ABAGAIL package
2. KColorEvaluationFunction.java
3. KHillsEvaluationFunction.java
4. KnapsackEvaluationFunction.java
5. neuralnetwork.py
6. car.data
7. khills.py
8. kcolors.py
9. knapsack.py
10. knapsack_statistics
11. kcolors_statistics