Implementation of randomized optimizations algorithms : 1

Analysis of Credit Rating dataset: 1

Data set : 1

Software to run the Algo : 1

A. Applying Genetic Algorithm to CreditRating dataset 1

B. Applying Randomized Hill Climbing to CreditRating dataset 2

C. Applying Simulated Annealing to CreditRating dataset 3

New Optimization Problem domains 4

Max K-Coloring Problem 4

N-Queen Problem 4

Cart Centering Problem 5

Job Scheduling Problem 6

Analysis of IonoSphere dataset (all continuous values) 6

Data set : 6

Software to run the Algo : 6

A. Applying Genetic Algo to the Ionosphere Dataset 6

B. Applying Simulated Annealing to IonoSphere dataset 8

C. Applying Random Hill Climbing Algo to IonoSphere dataset 9

## Implementation of randomized optimizations algorithms :

Random local search algo keeps track of only single current state and moves (randomly) only to neighboring state requiring little memory. It ignores possible combination of paths and can often find reasonable solutions in large or infinite continuous space. Random local search focuses on pure optimization problem where all states have an objective function and Goal is to find state with max (or min) objective value.

### Analysis of Credit Rating dataset:

#### Datasets :

* CreditRating-train.txt from the previous assignment
* File : randomized\_optimizations\_project/resources/ CreditRating-train.txt

#### Software to run the Application :

I have created Java Programs leveraging ABAGAIL

**project** : **randomized\_optimizations\_project/**

* com.ml.tests.optimizations.CreditRatingTest.java
* com.ml.tests.optimizations.NQueensTest.java
* com.ml.tests.optimizations.MaxKColoringTest.java
* com.ml.tests.optimizations.JobSchedulingTest.java
* com.ml.tests.optimizations.CartCenteringTest.java

**Result of Neural Network Algo as performed in assignment-1 using Gradient Descent in Back-propagation Network**

In Assignment-1 offered accuracy 99.8% while running over training samples. But ANN could correctly classify only 40% test instances.

* Performance improves a bit (64% accuracy rate against test data) when we specify a slower learning rate (0.2)
* It takes really long time ( 5.4 s) to build the model.
* Even with a stopping condition like validationThreshhold , trainingTime, validationSetSize accuracy rate does not improve.
* Interestingly if we decrease momentum , lower the learning rate and increase hidden layers then performance improves. For example :
  + If learning rate = 0.12 , momentum = 0.01 , #hidden layers = 6, then accuracy is 72%
  + Also it builds the model in 0.19 second as the number of hidden layers have increased!
* Overall, MLP is not able to converge to a solution for this particular training set due to
  + either unbiased estimators but probably with high variance (due to over-fitting) ;
  + or a rigid model in contrast , leading to small variance but high bias.
  + It not only takes longer time but also runs the risk of getting caught into a local optima.

Next we shall see how we can perform random local search to avoid local optima.

**Result of Applying Randomized optimization instead of Gradient descent in Back-Propagation Network**

* randomized\_optimizations\_project/
* java -cp randomized\_optimizations.jar:lib/ABAGAIL.jar com.ml.tests.optimizations.CreditRatingTest
* Or inside Eclipse IDE , **import** rand\_optimization\_tests project and

**run** rand\_optimization\_tests/src/com/ml/tests/optimizations/ CreditRatingTest.java

(one can change the various params of the algo in the code)

…………..

**for**(**int** i = 0; i < *oa*.length; i++) {

*networks*[i] = *factory*.createClassificationNetwork(

**new** **int**[] {*inputLayer*, *hiddenLayer*, *outputLayer*});

*nnop*[i] = **new** NeuralNetworkOptimizationProblem(*set*,

*networks*[i], *measure*);

}

*oa*[0] = **new** RandomizedHillClimbing(*nnop*[0]);

*oa*[1] = **new** SimulatedAnnealing(1E11, .1, *nnop*[1]);

*oa*[2] = **new** StandardGeneticAlgorithm(100, 50, 10, *nnop*[2]);

#### A. Applying Genetic Algorithm to CreditRating dataset

The genetic algorithm (GA) is an optimization and search technique based on the principles of genetics and natural selection. A GA allows a population composed of many individuals to evolve under specified selection rules to a state that maximizes the “fitness” (i.e., minimizes the cost function).

**Pros :**

• Doesn’t require derivative information,

• Simultaneously searches from a wide sampling of the cost surface,

• Deals with a large number of variables,

• Is well suited for parallel computing,

• Optimizes variables with extremely complex cost surfaces (they can jump out of a local minimum),

• Provides a list of optimum variables, not just a single solution,

* Random exploration (via CrossOver) can find solutions that other local search algo can’t find

**Cons :**

* In some cases, calculus-based derivatives to find a solution of a well-behaved convex function can outperform Genetic Algorithm as GA could be still spending cycles to analyze the costs of the initial population.
* Also it needs to come up with more bits for real-valued data.
* We need to be aware that if GA converges too soon into one region of cost surface, it may be caught into a local minima ! One solution is to randomly introduce mutations !

GA takes longer time with a sample size 200, mating nodes to 100, nodes to mutate 10.

Correctly classified 555.0 instances.

Incorrectly classified 20.0 instances.

Percent correctly classified: 96.522%

Training time: 2.993 seconds

Testing time: 0.002 seconds

GA works faster if we set population size to 100, mating nodes to 50, nodes to mutate 20.

Correctly classified 555.0 instances.

Incorrectly classified 20.0 instances.

Percent correctly classified: 96.522%

Training time: 1.633 seconds

Testing time: 0.002 seconds

#### B. Applying Randomized Hill Climbing to CreditRating dataset

It is “a loop that continuously moves towards increasing value”. It can randomly choose among the set of best successors to avoid local minima.

The main benefit of Hill Climbing search is that it requires only a limited memory.

We need to select a local change that should improve the current value of Objective Function.

Like gradient descent algorithms in continuous spaces it approaches a local minima but instead of using the gradient, hill climbing uses random local search. HA differs from Gradient Descent algo which adjust all of the values in random variable X at each iteration according to the gradient of the hill.

The neural networks are trained in batch mode where the performance is measured by the root mean squared error (rmse) over the training set. Whereas NW is a new weight matrix (`the mutant') which is obtained by adding a randomly generated step matrix dNW to W

HC uses random numbers for

1. Generation of the initial weight matrix W.

2. Selection of a random number which determines the variable stepsize.

3. Calculation of the step matrix dNW whose components are all new random numbers.

We observe that if number of iterations is 10 , RHC does not perform well !

It gets stuck at local optima .

Correctly classified 20.0 instances.

Incorrectly classified 555.0 instances.

Percent correctly classified: **3.478%**

Training time: 0.148 seconds

Testing time: 0.012 seconds

Once the number of iterations is 100, RHC correctly classified 549.0 instances as it can better explore more neighbors and move past ridges and plateaus .

Incorrectly classified 26.0 instances.

Percent correctly classified: 95.478%

Training time: 0.346 seconds

Testing time: 0.003 seconds

#### C. Applying Simulated Annealing to CreditRating dataset

Simulated Annealing :

It is an example of random local search method to incorporate sideways and downhill moves at the beginning with a probability based on the size of the change in objective function. The idea is to do enough exploration of the whole space early on so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, a plateau, or a ridge.”

1. Let X := initial config

2. Let E := Eval(X)

3. Let i = random move from the moveset

4. Let Ei := Eval(move(X,i))

5. If E < Ei then X := move(X,i)

E := Ei Else with some probability,

accept the move even though things get worse:

X := move(X,i)

E := Ei

6. Goto 3 unless reached end of iteration.

If we decrease the temperature by a factor of 0.99 , and number of iterations is ~ 50 , SA does not perform at all .

Whereas within same number of iterations GA and RHC perform quite well !

Correctly classified 35.0 instances.

Incorrectly classified 540.0 instances.

Percent correctly classified: 6.087%

Training time: 0.091 seconds

Testing time: 0.003 seconds

If we increase the number of iterations to 500 with temp reduction rate 0.99, still SA performs poorly (though better than last time)

Correctly classified 151.0 instances.

Incorrectly classified 424.0 instances.

Percent correctly classified: 26.261%

Training time: 0.769 seconds

Testing time: 0.002 seconds

Finally if the temperature is decreased by 0.1 (much more slowly) and number of iterations is 10, **the performance of SA improves a lot** . The main reason for improvement of performance is the ‘frequency of moves to explore non-optimal paths and the size of such non-optimal space (plateau) decreases’ as we lower the temperature.

Results for SA:

Correctly classified 518.0 instances.

Incorrectly classified 57.0 instances.

Percent correctly classified: 90.087%

Training time: 0.046 seconds

Testing time: 0.005 seconds

If the temperature is decreased by 0.1 and number of iterations is 500, the performance of SA further improves.

Results for SA:

Correctly classified 555.0 instances.

Incorrectly classified 20.0 instances.

Percent correctly classified: 96.522%

Training time: 0.894 seconds

Testing time: 0.002 seconds

### New Optimization Problem domains

So far we have analyzed GA, SA, RHC . Lets now also apply MIMIC to some interesting problems.

MIMIC uses prior solutions to build a model of the solution space that focuses on areas of the space that are likely to contain high performing candidate solutions

#### Max K-Coloring Problem

A graph is K-Colorable if it is possible to assign one of k colors to each of the nodes of the graph such that no adjacent nodes have the same color.

First we initialize a sample graph with a set of colors such that adjacent nodes of any node have unique colors.

The fitness function tries to find if a combination of k-colors can be assigned to the Graph and how many iterations does it take to take the decision.

java -cp randomized\_optimizations.jar:lib/ABAGAIL.jar com.ml.tests.optimizations.MaxKColoringTest

We see that even with large number of iterations; RHC and SA fails to find a working combination of Max-K colors.

GA performs better because *any* crossover point is representative of some of the underlying structure of the graphs used.

MIMIC is well suited for problems where the computation of Cost function is expensive so that as much information as possible can be extracted from each iteration in order to best focus our search. It efficiently communicates information about the cost function obtained from one iteration of the search to later iterations of the search.

MIMIC begins by generating a random population of candidates chosen uniformly from the input space. From this population the median fitness is extracted and is denoted θ0. The algorithm then proceeds:

1. Update the parameters of the density estimator of pθi (x) from a sample.
2. Generate more samples from the distribution pθi (x). 3.
3. Set θi+1 equal to the Nth percentile of the data. Retain only the points less than θi+1. Repeat.

Number of vertices = 50, Adjacent nodes per vertex = 4, Possible K colors = 8

RHC: 340

Failed to find Max-K Color combination !

Time : 89

============================

SA: 340

Failed to find Max-K Color combination !

Time : 72

============================

GA: 350

Found Max-K Color Combination !

Time : 28

============================

MIMIC: 350

Found Max-K Color Combination !

Time : 41

#### N-Queen Problem

Place N queens on a N\*N chess board so that no queen is attacking another. Each queen must be in one of the N columns.

The Fitness function finds the number of steps needed to find first solution (first board position of non-attacking pairs of queens).

java -cp randomized\_optimizations.jar:lib/ABAGAIL.jar com.ml.tests.optimizations.NQueensTest

For N = 10

RHC: 44

Time : 0

============================

SA: 45

Time : 0

============================

GA: 42

Time : 62

============================

MIMIC: 42

Time : 60

RHC and SA takes 0 ms . GA takes 60 ms and MIMIC takes 180 ms for the same number of iterations .

If we decrease the #iterations for MIMIC to 5 and t#tokeep to 10, then we get the same result but in much faster time (30 ms)

If we decrease the #values\_to\_mutate then GA takes lesser time.

MIMIC performs better because it is able to capture all of the structural regularity within the inputs.

RHC: 44.0

Time : 1

============================

SA: 45.0

Time : 0

============================

GA: 42.0

Time : 34

============================

MIMIC: 42.0

Time : 30

Even though MIMIC needs smaller number of iterations , its much slower than RHC and SA.

**Discrete Space Hill Climbing Algorithm**

currentNode = startNode;

loop do

L = NEIGHBORS(currentNode);

nextEval = -INF;

nextNode = NULL;

for all x in L

if (EVAL(x) > nextEval)

nextNode = x;

nextEval = EVAL(x);

if nextEval <= EVAL(currentNode)

//Return current node since no better neighbors exist

return currentNode;

currentNode = nextNode;

#### Cart Centering

Your only means of controlling the cart are to fire your thrusters forwards or backwards. You want to stop the cart at the center of the path as quickly as possible. For the sake of brevity we are considering one variable i.e. Velocity.

Fitness for the cart centering problem states that if the velocity is close to zero within the margin of error , then it’s a success.  Some randomly generated strategies may simply be to accelerate forward indefinitely, which would certainly never bring the cart to rest at the center. For this reason, we need a timeout value, usually 10 seconds. If an individual has not centered the cart in under 10 seconds, it is simply assigned a time of 10 seconds for that trial.

java -cp randomized\_optimizations.jar:lib/ABAGAIL.jar com.ml.tests.optimizations.CartCenteringTest

For 20 discrete velocity values ( between 0 -10) , the cart comes to the center after 869 movements !

RHC, SA, GA , MIMIC all offer similar result !

As usual SA is the fastest and MIMIC is the slowest algo.

RHC: 209.0

Time : 12

============================

SA: 209.0

Time : 8

============================

GA: 209.0

Time : 89

============================

MIMIC: 209.0

Time : 101

#### Job Scheduling

Say each of 3 machines performs same set of 5 jobs with different amount of time.

We sum up the job times of each machine.

Then the Goal is to minimize the makespan i.e. to find which machine yields minimum makespan.

Here we see Simulated Annealing performs faster than all other algo while finding the minimum makespan.

RHC: 2

Time : 4

============================

SA: 1

Time : 0

============================

GA: 16

Time : 15

============================

MIMIC: 13

Time : 38

### Analysis of IonoSphere dataset (all continuous values)

We apply the SHC, SA and GP algo using Weka as it allows us (i) to tune quite a few params , (ii) understand better how the algo works , (3) provides a nice error chart .

#### Data set :

Ionosphere data used in assignment 1

Files : ionosphere\_train.arff (citation : <https://archive.ics.uci.edu/ml/datasets/Ionosphere> )

#### Software to run the Algo :

<http://sourceforge.net/projects/wekagp/files/latest/download>

#### A. Applying Genetic Algo to the Ionosphere Dataset

*For continuous valued inputs, formulate the cost function*

*Generate initial population with random numbers*

*Find cost of each chromosome*

*Select mates (After a Z% selection rate , apply pairing and mating process ) [ z= max allowed cost]*

*Mating*

*Mutation*

*Convergence check*

*Again find cost for each chromosome*

*Done*

*Say cost = f (x, y) = ax sin(bx) + cy sin(dy) Subject to the constraints: 0 <= x <= M and 0 <= y <= N*

*chromosome = [x, y] .*

*Our goal is to find the global minimum value of f(x, y)*

We no longer need to consider how many bits are necessary to accurately represent a value. Instead, x and y have continuous values that fall between the bounds.

Scheme: weka.classifiers.functions.GeneticProgramming

Relation: ionosphere

Instances: 315

Attributes: 35

Test mode: user supplied test set: 36 instances

* One Class Weight Evaluator : returns confidence (0.0 to 1.0) of instance being of current class

Tree Population (ramped) Half and Half Initializer : initializes a population of program trees using the Full and Grow method, creating trees with depth from 2 to maximum depth, with an equal number of programs for each depth.

* Fitness-proportionnal Selector : selects programs with probability based on fitness (with fitness / sumFitness probability for each program).
* Keep Bests Elite Manager : keeps 1 best programs since beginning of run in memory.
* Select After Evolution Controller : we create a new population of 100 children programs. From the total of parents and the children is selected enough individual to produce the next generation of programs.
* Tree Crossover Operator : "crosses" two programs together, switching a random sub-tree from each of the programs. Results are two offsprings.
  + Proportion = 0.9
  + Number of parents = 2
  + Number of children = 2
* Tree Mutation Operator : mutates a whole randomly selected sub-tree from a program.
  + Proportion = 0.07
  + Number of parents = 1
  + Number of children = 1
* New Program Tree Operator : creates completely new programs.
  + Proportion = 0.03
  + Number of parents = 1
  + Number of children = 1

Target fitness = 0.9

Maximum time = 0.033 minutes

Max generations = 20

=== Evaluation on training set ===

**Time taken to build model**: 3.94 seconds

Correctly Classified Instances 291 92.381 %

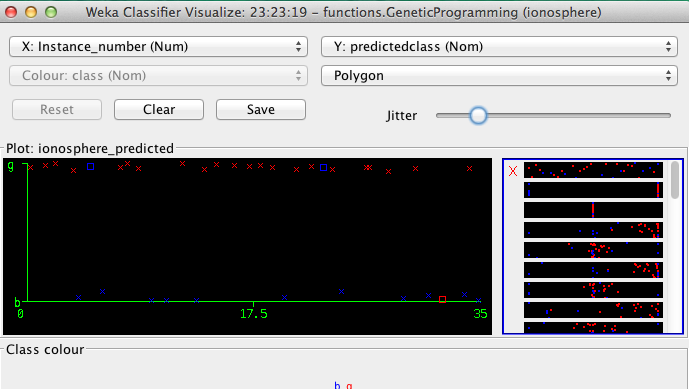
Incorrectly Classified Instances 24 7.619 %

=== **Evaluation on test set** ===

**Time taken to build model**: 3.5 seconds

Correctly Classified Instances 33 91.6667 %

Incorrectly Classified Instances 3 8.3333 %



#### B. Applying Simulated Annealing to IonoSphere dataset

Initially SA is set to decrease the temperature by 0.999 with a start temperature 10 and total iterations 1000

Its takes long time and does not respond at all.

As soon as we reduce the starting temperature to 2.0 , we get a good result in a less time.

=== Evaluation on training set ===

Time taken to build model: 0.32 seconds

Correctly Classified Instances 305 96.8254 %

Incorrectly Classified Instances 10 3.1746 %

**=== Evaluation on test set ===**

Time taken to build model: 0.32 seconds

Correctly Classified Instances 33 91.6667 %

Incorrectly Classified Instances 3 8.3333 %

The test result **further improves** if we set starting temperature as 2.0 and decrease it by 0.1 in 1000 iterations.

**=== Evaluation on test set ===**

Time taken to build model: 0.85 seconds

Correctly Classified Instances 34 94.4444 %

Incorrectly Classified Instances 2 5.5556 %

Interestingly, if we increase the starting temperature to 10 with a rate of decrease 0.1, we get an accuracy of 98.1% (on the training set) at the cost of longer durations ( 6 seconds ) .

=== Detailed Accuracy By Class ===

TP Rate FP Rate Precision Recall F-Measure Class

0.846 0 1 0.846 0.917 b

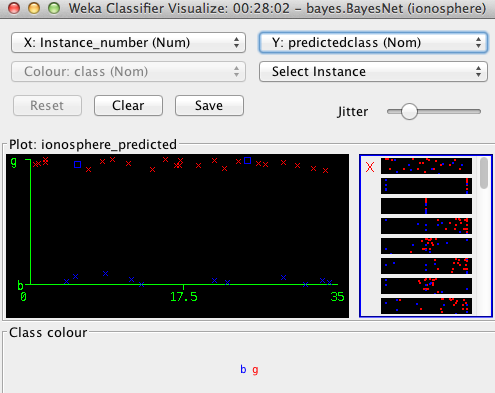
1 0.154 0.92 1 0.958 g

=== Confusion Matrix ===

a b <-- classified as

11 2 | a = b

0 23 | b = g



**=== Evaluation on training set ===**

Correctly Classified Instances 309 98.0952 %

Incorrectly Classified Instances 6 1.9048 %

If the temperature was increased SA could not improve on the performance of HC and the convergence of the networks was delayed or inhibited.

#### C. Applying Random Hill Climbing Algo to IonoSphere dataset

Hill Climber Algo initialized with Naïve Bayes network and Markov Blanket correction is applied. We use.

We get a consistent good result comparable with SA (similar performance of RHC irrespective of Bayes / MDL / Entropy to judge the quality of the network structure)

A lower alpha (used to estimate probability table for simple estimator) does not have any impact on the accuracy.

**Continuous Space Hill Climbing Algorithm**

currentPoint = initialPoint; // the zero-magnitude vector is common stepSize = initialStepSizes; // a vector of all 1's is common

acceleration = someAcceleration; // a value such as 1.2 is common

candidate[0] = -acceleration;

candidate[1] = -1 / acceleration;

candidate[2] = 0; candidate[3] = 1 / acceleration;

candidate[4] = acceleration;

loop do

before = EVAL(currentPoint);

for each element i in currentPoint do

best = -1;

bestScore = -INF;

for j from 0 to 4 // try each of 5 candidate locations

currentPoint[i] = currentPoint[i] + stepSize[i] \* candidate[j];

temp = EVAL(currentPoint);

currentPoint[i] = currentPoint[i] - stepSize[i] \* candidate[j];

if(temp > bestScore)

bestScore = temp;

best = j;

if candidate[best] is not 0

currentPoint[i] = currentPoint[i]+stepSize[i]\* candidate[best];

stepSize[i] = stepSize[i] \* candidate[best]; // accelerate

if (EVAL(currentPoint) - before) < epsilon

return currentPoint;

=== Evaluation on test set ===

Time taken to build model: 0.02 seconds

Correctly Classified Instances 33 91.6667 %

Incorrectly Classified Instances 3 8.3333 %

I would prefer RHC to SA as RHC offers the same accuracy with 90% less time !

=== Detailed Accuracy By Class ===

TP\_Rate FP\_Rate Precision Recall F-Measure Class

0.769 0 1 0.769 0.87 b

1 0.231 0.885 1 0.939 g

=== Confusion Matrix ===

a b <-- classified as

10 3 | a = b

1. 23 | b = g

The following graph clearly depicts the confusion matrix , 23 samples correctly classified as g ( True Positive = 1 for class ‘g’) and 3 ‘b’ class samples misclassified as ‘g’ .

