Project 7 : Genetic Algorithms

Joshua Bodah

ML F12

Overview

In this paper, a genetic algorithm was used to learn a neural network on census income data. Most code was written from scratch. Experiments were run to compare.

Data Sets

- *Adult:*

  Uses census data to predict whether an individual will make more or less than $50,000/yr

  - 32561 instances
  - 6 numeric attributes, 8 nominal attributes, 1 class attribute
  - Contains 4262 missing attributes
  - **24720 (75.9%) <=50K, 7841 (24.1%) >50K**

Preprocessing of the Data

Weka was used to convert the nominal attributes into binary numeric attributes. Code was written to normalize any numeric attributes as well as to remove any instances with missing values.

Algorithms and Code

A genetic algorithm was written from scratch in Ruby. The code can be found at: http://web.cs.wpi.edu/~jbodah/code. The algorithm was based on the psuedo-code found in Russel and Norvig [1]. The implementation is composed of three major classes for running the code (Individual, Population, and GeneticAlgorithm) and several support class for parsing and manipulating data. Individual and Population both extend Ruby's Array class and provide methods for mutation, crossover, and determining fitness. One modification was made to use arrays in place of binary strings to encode the individuals. The implementation of mutation and initialization of these individuals was therefore left to the specific subclass of Individual. The genetic algorithm written was used to learn a neural network. The details of the design decisions can be found below:

- *Hypothesis encoding:* The neural network was encoded via an array of weights corresponding to each node. That is, if there were 2 nodes, the first $n + 1$ weights (including w_0) corresponded to the first node, the next $n + 1$ to the second node, etc. Therefore, the number of nodes for the neural network had to be passed in as a parameter.

- *Size and initialization of population:* The size of the population varied between experiments. Weights for each individual in the population were initialized using a random number between [-1,1].

- *Fitness function:* The classification accuracy of a neural network was used as the fitness benchmark:
    - positive = 0
    - For each data_instance
        - sum = 0
        - For each node

            if $\sum_{i=0}^{n} x_i w_i > 0$, then sum += 1

            else, sum += -1
        - classification = (sum > 0) ? CLASS_A : CLASS_B
        - positive += 1 if classification == actual_class(data_instance)
    - return positive/size(data_set)

- *Selection method:* Fitness proportionate selection was used. During each reproduction cycle, parents were randomly sampled with replacement based on their fitness. Each pair of parents produced two children. *n*/2 pairs of parents were chosen for reproduction to retain the population size.

- *Crossover method:* Single point crossover was implemented by selecting a random crossover point in the chromosome and giving one child the [0, crossover_point – 1] elements of one parent and the [crossover_point, size(parent)] elements of the other. Another child was then produced using the same crossover point but with the parents switching bit positions. By using an array instead of a binary string, we gained flexibility in how information is encoded in a chromosome while retaining the simplicity of crossover using binary strings (as opposed to making sure that the crossover never splits a binary string in the middle of some value).

- *Mutation method:* When a child was produced, it had a random chance of being mutated. This mutation rate could be set as a parameter. When mutation took place, a random element was selected, and that weight was randomly regenerated.

- *Termination criteria:* Two termination conditions were implemented. The first, which was mandatory, was a fitness threshold. The search would stop if the fitness of any individual ever exceed this threshold. The second, which was optional, was a timeout condition to allow time-limited searches.

Objectives of the Data Mining Experiments

From the data mining experiments, we hope to:

1. Compare learning a neural network through back-propagation with learning one via a genetic algorithm
2. Find the population size which yields the best network for the “adult” data set in a time-limited environment
3. Analyze the convergence of the population fitness of genetic algorithms with respect to various parameters
4. (Advanced topic) Attempt to improve performance of the genetic algorithm through concurrency

Performance Metrics

The two performance metrics that will be considered are time to find a solution and classification accuracy.

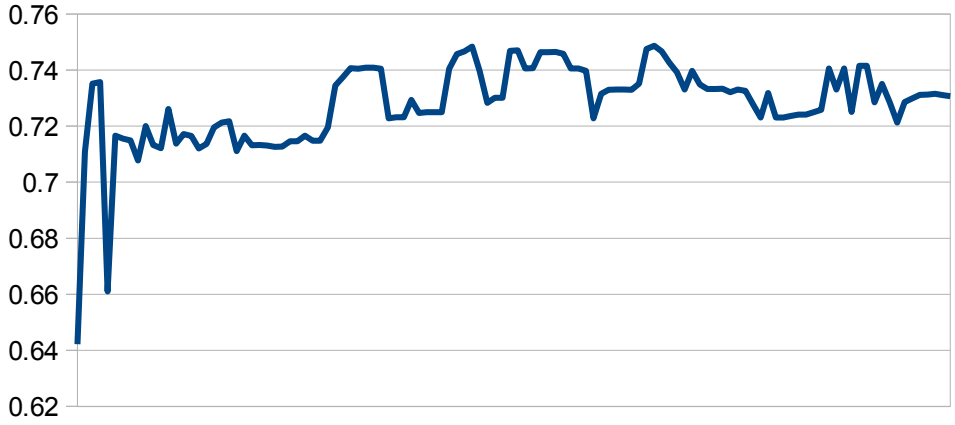
Experiments

Comparison of genetic algorithm and back-propagation

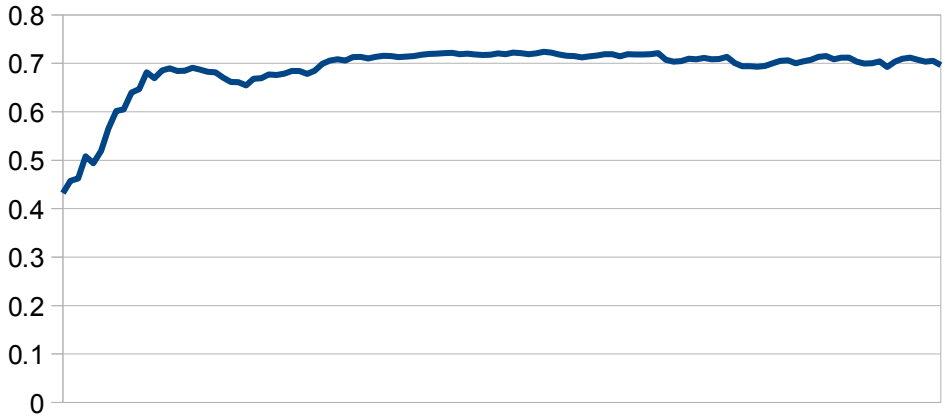
Training Weka's MultilayerPerceptron to fit a model to the entire data set (that is, trained and tested on the entire data set) took about one minute to train a neural network with 84.6% accuracy using one hidden layer node. The goal of this experiment was to use a genetic algorithm to find a comparable solution.

The genetic algorithm was run for 4.5 hours with a population size of 20 and mutation rate of 0.3. The best solution found within this time-frame had a classification accuracy of about 74%, a significantly worse result than that of back-propagation. Below are graphs of the log data. We can see that the algorithm finds an answer which is about as good as the best fairly early on in the run-time but seems to move away from it and consequently performing worse. This could be a result of using fitness proportionate selection instead of a deterministic approach such as tournament selection. The population's average fitness and the standard deviation between the fitnesses decreases and then stablizes. This could mean that the solutions have converged or that the population size is too small. Using a random restart approach could help to avoid situations where the individuals all converge to similar models.

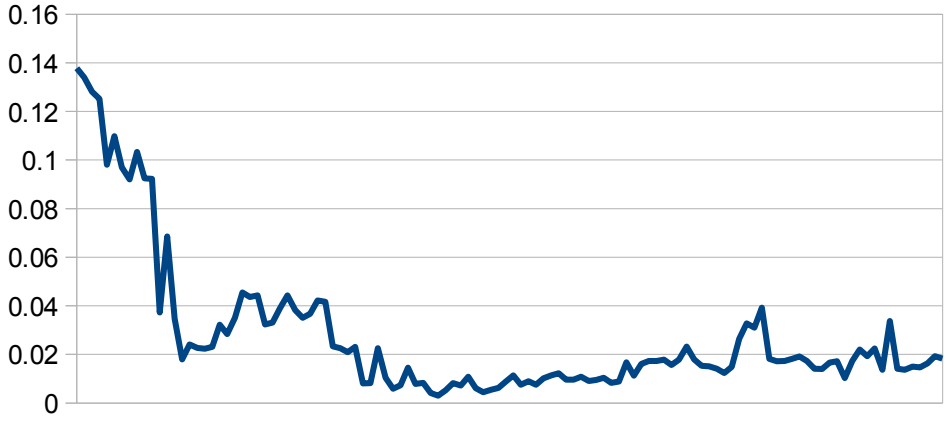
Population Max Fitness Over Time



Population Fitness Average Over Time



Population Fitness Standard Deviation Over Time



Best population size in a time-limited environment

For this experiment three trials of the time-limited genetic search were run using an hour as the threshold and a mutation rate of 0.3. The population size for each of the trials was 20, 50, and 100 respectively.

Pop_size	Production cycles	Initial population best	Initial population average	Initial population std_dev	Final population best	Final population average	Final population std_dev
20	27	75%	56%	0.14	75%	71%	0.04
50	9	75%	50%	0.16	75%	67%	0.08
100	4	75%	51%	0.15	77%	62%	0.13

From the above results we can gather that each of the trials ultimately produced models with similar accuracy. Using a larger population seems to have the advantage of having a more diverse field of candidate parents. This increases the chances of getting a good parent to reproduce with the rest of the data set and would likely be the best solution in many cases. For the “adult” data set, it seems wise to choose a large population size.

Effects of concurrency on genetic algorithm performance

Here, the genetic approach was modified to use multiple threads. Since MRI (the standard Ruby interpreter) does not support parallel threads, JRuby had to be used instead. New threads were spawned for both reproduction and determining an individual's fitness. Two major experiments were run: one time-limited trial to compare the multithreaded approach to the single threaded one and another which was thresholded only by classification accuracy. The latter experiment was able to finish between 3-4 hours but ran into an exception shortly afterwards before the model could be logged. I haven't been able to reproduce the model yet.

For the time limited experiment, trials were run on a quad-core machine. Population sizes of 24, 50, 74, and 100 were used, and the trials lasted 5 minutes. From the trials we see that using parallel threads allows the program to process data 3-4 times faster than the single-threaded approach. Given that the machine I used is a quad core, this result makes sense intuitively.

Parallel?	Pop_size	Individuals processed	Population best	Population average	Population std_dev
N	24	284	75%	72%	0.04
N	50	222	76%	60%	0.11
N	74	198	75%	58%	0.15
N	100	252	75%	54%	0.15
Y	24	793	75%	74%	0.01
Y	50	700	75%	68%	0.07
Y	74	741	75%	66%	0.10
Y	100	800	75%	59%	0.06

Appendix: Code

What follows is the code for the parallel implementation of the genetic algorithm. All of the code written can be found at http://web.cs.wpi.edu/~jbodah/code.

```
# parallel_genetic_algorithm.rb

require 'thread'

class ParallelGeneticAlgorithm < GeneticAlgorithm
  # Return an individual whose fitness exceeds the threshold
  def search(threshold)
    mutex = Mutex.new

    until @population.parallel_contains_an_individual_with_fitness_as_good_as?
(threshold)
      # DEBUG CODE
      puts "Reproducing... #{@population.size}"
      new_population = @population.class.new
      threads = []

      for index in (0...(@population.size/2)) do
        threads << Thread.new {
          x = @population.random_selection
          y = @population.random_selection
          children = @population.first.class.reproduce(x, y)
          children.each do |child|
            child.mutate if rand() <= mutation_rate
          end
          mutex.synchronize do
            children.each {|child| new_population.push(child)}
          end
        }
      end
      threads.each do |thread|
        thread.join
      end

      @population = new_population
    end
    return @population.best_individual
  end
end
```

Appendix: Code (cont)

```
# population.rb

class Population < Array
  attr_accessor :best_individual

...

  # Used to terminate search
  # Assumes threshold in [0,1] and assumes binary classification
  def parallel_contains_an_individual_with_fitness_as_good_as?(threshold)
    # Multithread
    threads = []
    self.each do |individual|
      threads << Thread.new(individual) do |indiv|
        indiv.calculate_fitness
      end
    end

    threads.each do |thread|
      thread.join
    end

    self.each do |individual|
      fitness = individual.fitness
      # DEBUG CODE
      puts fitness
      if fitness >= threshold
        @best_individual = individual
        return true
      end
    end
    false
  end

...

end
```

Appendix: References

[1] Russel, Stuart and Norvig, Peter, 2010, Beyond Classical Search: Genetic Algorithms, *Artificial Intelligence: A Modern Approach*, Pearson, Upper Saddle River, p.129