Genetic Algorithms

J.R. Powers-Luhn

***Abstract*—Genetic algorithms are an approach to optimization which combines a random set of starting solutions combined with a method of iteratively creating new generations of potential chromosomes from the high-performing solutions in the current generation. The performance of these algorithms for a selection of parameters is examined and compared with a gradient descent optimization. The accuracy of the results are comparable, despite significant differences in the result vectors.**

I. Introduction

GENETIC algorithms are an approach to optimization that takes its inspiration from biology. Rather than calculating a gradient and using that information to iteratively move towards a minimum value on an error surface, the fitness of several candidate solutions is evaluated. This population of potential solutions (referred to as chromosomes) is then ranked based on the value of a fitness function and a new population is generated via a process of:

- preserving high-fitness chromosomes,
- combining elements from high-fitness chromosomes to form new chromosomes, and
- ”mutating” chromosomes–randomly altering genes from existing chromosomes.

In order to begin this process, a population of candidate solutions to the problem (called ”chromosomes”) are generated. The number of chromosomes generated is a trade off between the ability to explore the solution space completely and the time it takes for the system to converge on an answer. Each chromosome has either the parameters to be optimized exactly or approximated as a series of binary bits. The fitness of each candidate is evaluated (with the intermediate step of converting into decimal values in the case of binary.

The chromosomes are then ranked and sorted into three groups. Chromosomes with the best fitness scores are designated as ”elite candidates”, who are propagated through to the next generation without mutation (this will include the best solution from the current population, preventing the solution from getting worse between generations). Another group are selected to be ”parents”, and are used to generate the remaining chromosomes required to bring the population back up to the previous level. Finally, some chromosomes may not meet a minimum fitness threshold and are rejected without being added to the breeding pool. The size of each category is a design decision that trades time to converge on an answer against complete search of the problem space.

The chromosomes that make up the subsequent generation are then generated. Elite chromosomes are propagated directly without alteration. Remaining slots are filled by ”breeding” other candidates together by combining their parameters via a variety of means–each parameter may be selected from one of the two parents at random, possibly weighted by the parent’s fitness, or a modified average of each parameter may be used, or a crossover technique where the first n elements come from one parent and the last m elements from the other. Whichever method is selected should allow for a full search of the parameter space.

This evaluation step is then repeated as necessary until some minimum fitness is reached or the algorithm converges.

In this report we sought to optimize parameter selection for a local ridge regression problem, attempting to minimize equation 1:

$$\left(\overline{\overline{A}}^T\overline{\overline{A}}+\overline{\overline{\alpha^2}}\right)^{-1}A^T\vec{b}, \qquad (1)$$

where $\overline{\overline{A}}$ represents the inputs and $\vec{b}$ represents the output of a data set.

II. Methodology

The data set examined consisted of 5000 records of forty-three input values used to predict a single output value. First the data was imported and split into training (2000 records), testing (1500 records), and validation sets (1500 records). The data were scaled such that the training set had a zero mean and unit variance for each variable. A cross validation approach was used where the optimization continued using the training data until the testing data RMSE fell below a threshold value. The accuracy of the resulting solution was then evaluated using the validation data.

To provide a reference for comparison, a gradient descent optimization was used to find the weight parameters $\vec{\alpha}$. The search was initialized based on the correlation coefficients of the principle component loadings and the scaled training set output.

Several variants of the genetic algorithm were examined, including method for parent selection (stochastic uniform, roulette, and tournament), mutation rate (1% or 25% of genes), and number of elite candidates maintained from the previous generation (10 or 100). These parameters were varied widely in an attempt to determine if they impacted the number of generations required to converge or the performance on the validation set. Only continuously-valued parameters were examined.

Three parent selection methods were employed for this paper. The first, stochastic uniform, selected parents for the next generation by first placing the chromosomes on a number line, sized in proportion to their current fitness (inverse fitness in this case, since it is a minimization problem). A randomized first step, followed by steps of fixed size after that were used to select the parents from this number line. This had the effect of favoring more fit parents while effectively randomizing the less fit parent. The roulette strategy was somewhat more random–it assigned weighted probabilities to each parent based on their fitness values and selected at random. It was therefore less biased towards fit parents. The final parent selection method was the tournament method, where four possible parents were selected in each round, with the most fit of these four added to the pool of parents. The parent chromosomes were bred by selecting each gene randomly from one parent or the other with no weighting for fitness.

Two mutation rates were examined in order to determine the impact on the solution. Mutations were accomplished by selecting a fraction of the genes from the non-elite population and adding a Gaussian noise value to that gene. Genetic algorithms with the standard deviation of the Gaussian noise set to 1 and 10 were examined.

The number of ”elite candidates” was examined at 5 and 50% of the population. It was expected that this would affect the number of generations to converge or the root mean squared error of the final candidate.

The range of algorithm settings is shown in table I.

III. Results

As expected, the local ridge regression showed filter factors mostly corresponding to the highly correlated parameters in the input, with one uncorrelated parameter still having a filter factor above 0.2.

The genetic algorithms examined tended to have low α values, resulting in higher filter factors than the local ridge regression (figure 1). Since the gradient descent method found lower filter factors in all cases (but especially in the range $\alpha_1 - \alpha_{30}$ it was theorized that the genetic algorithm was not exploring the space adequately. Increasing the mutation rate, which should improve this, was not effective in correcting this discrepancy, however.

The validation set was used to determine the accuracy of each fitting method used. Gradient descent resulted in a root mean squared error of 0.236 for the validation set. It was expected that the higher filter factors would allow more over fitting of the validation data, resulting in a higher RMSE for the genetic algorithms. However, the actual errors for the four genetic algorithms were very close to each other and lower than the gradient descent method at 0.229.

Altering the fraction of the population considered to be elite did not affect either the accuracy of the solution or the time required to converge to within error. All of the algorithm settings caused the fittest member of the population to converge within 58-61 generations.

TABLE I
Genetic algorithm parameters

GA#	Parent	Mutation	Iterations	Elite	RMSE
1	Stochastic uniform	1%	59	5%	0.229
2	Roulette	1%	58	5%	0.229
3	Roulette	1%	61	50%	0.229
4	Tournament	25%	60	10%	0.229

All in all the genetic algorithms performed about as well as the gradient descent algorithm in delivering local ridge regression coefficients, as measured by not over fitting the validation data. The plots of genetic algorithm 1 and the gradient descent closely match each other with neither over fitting noise (figure 2).

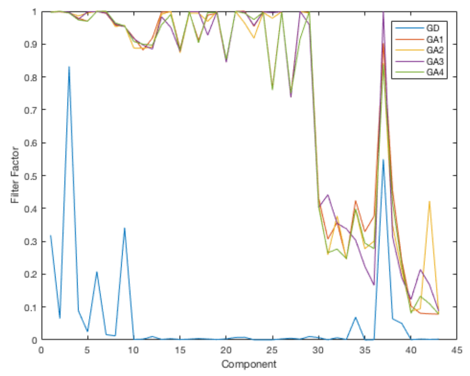

Fig. 1. Filter factor for gradient descent and various genetic algorithms

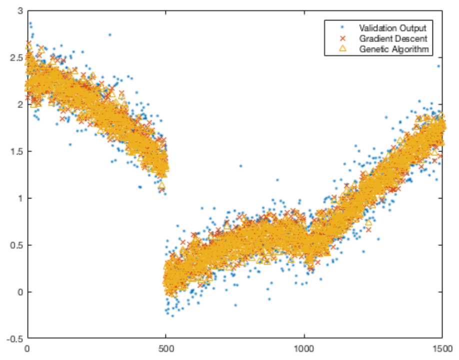

Fig. 2. Performance of gradient descent and genetic algorithm on validation set

IV. Conclusions

Genetic algorithms in a variety of configurations were used to fit a local ridge regression problem. The accuracy of the results was comparable to a gradient descent fit of the same data, even while the genetic algorithms produced markedly different filter factors. It is possible that these filter factors diverge but do not carry a high penalty in terms of accuracy in the test data set, meaning that there is no selection pressure to minimize the value of these elements. The larger filter factors may represent undesirable behavior, but this is not apparent in the validation set used here–it may be desirable to consider a different validation (or test) set to further improve the results of this method. The algorithms employed here are representative but by no means do they span all the variations that exist on the theme of genetic algorithms. Further work is called for to confirm that the available space has been effectively searched.

Appendix
Matlab code

Matlab code used to compare local ridge regression and genetic algorithm is included as an appendix.

```
% clear

%% Load the data

load sim.mat
train = Data([1:500 1501:2000 3001:3500 4501:end],:);
test = Data([501:1000 2001:2500 3501:4000],:);
val = Data([1001:1500 2501:3000 4001:4500],:);

x_train = train(:,[1:34 36:end]);
y_train = train(:,35);
x_test = test(:,[1:34 36:end]);
y_test = test(:,35);
x_val = val(:,[1:34 36:end]);
y_val = val(:,35);

%% Scale our data sets
[xs_train, x_mean, x_std] = zscore1(x_train);
[ys_train, y_mean, y_std] = zscore1(y_train);

% Scale the validation and test data to the same scale
xs_test = zscore1(x_test, x_mean, x_std);
ys_test = zscore1(y_test, y_mean, y_std);

xs_val = zscore1(x_val, x_mean, x_std);
ys_val = zscore1(y_val, y_mean, y_std);

%% Bound the alpha values
[loadings latent perExp] = pcacov(cov(xs_train));
scores = xs_train * loadings;
cc = corrcoef([scores ys_train]);
s = svd(xs_train);

% to start, back-calculate from the corellation coefficients
ff_start = abs(cc(1:end-1,end));
alpha0 = sqrt(s.^2 .* (1 - ff_start) ./ ff_start);

alpha_min = min(svd(xs_train));
alpha_max = max(svd(xs_train));
ff = @(alpha) s.^2 ./ (s.^2 + alpha.^2);

%% Initialize the alpha values at 20

%alpha0 = rand(size(xs_train,2), 1) * (alpha_max - alpha_min) + alpha_min
%alpha0 = 20 * ones(size(xs_train, 2), 1);

%% Fitness function and optimization settings
func = @(x) ridge_mse(x, xs_train, ys_train, xs_test, ys_test);
opt = optimset('plotfcn', {@optimplotx, @optimplotfval}, 'display', 'iter');

%% Gradient descent
[alpha_gd, fval, exitflag, output] = fminunc(func, alpha0, opt);
B_gd = (xs_train'*xs_train + diag(alpha_gd.^2))\(xs_train'*ys_train);
ys_v = xs_val * B_gd;
y_gd = ys_v * y_std + y_mean;

%% Genetic algorithm
% Default options
ga_opt1 = optimset('plotfcn', {@optimplotx, @optimplotfval}, 'display', 'iter');
```

```
alpha_ga1 = ga(func, 43)';
B_ga1 = (xs_train'*xs_train + diag(alpha_ga1.^2))\(xs_train'*ys_train);
ys_v = xs_val * B_ga1;
y_ga1 = ys_v * y_std + y_mean;

% Larger population
ga_opt2 = optimset('plotfcn', {@optimplotx, @optimplotfval}, 'display', 'iter');

%% Graphs and stuff!
f = figure;
plot(y_val, '.');
hold on;
plot(y_gd, 'x');
plot(y_ga1, '^');
legend('Validation Output', 'Gradient Descent', 'Genetic Algorithm');

g = figure;
plot(ff(alpha_gd));
hold on;
plot(ff(alpha_ga1));
ylabel('\alpha value');
legend('Gradient descent', 'Genetic algorithm');
title('\alpha values for optimization algorithms');

%% Calculate MSE for each
gd_err = sqrt(mean((y_gd - y_val).^2))
ga_err = sqrt(mean((y_ga - y_val).^2))
```