Chapter 4

Another Case Study: Genetic Algorithms

Genetic Algorithms

The section on Genetic Algorithms (GA) appears here because it is closely related to the problem of unsupervised learning. Much of what follows was done in collaboration with a student, Jenna Carr, who was working on her senior project.

It is probably easiest to think of a GA as a technique that will optimize some "fitness function", which is what we typically think of when we talk of biological evolution. Researchers use vocabulary that has its roots in biology, but we will try to keep the vocab neutral.

Let's start things off by considering how evolution works in its simplest form. We begin with some *population* that has *chromosomes*, and these chromosomes defines how well an individual matches its environment. By mating, chromosomes from different individuals are somehow combined to make new individuals, and a new generation appears.

Here are components that are common to almost all forms of a GA:

- A population of individuals, each with chromosomes.

  A key step is in how one translates a chromosome into a numerical string. We also need to decide what will constitute a valid string, or valid chromosome.

  For example, some practitioners will use binary strings. This works well for some problems, but in other problems we may want to use a continuum rather than a discrete set of points.

- A fitness function.

  This function uses chromosomes to measure how well an individual is "performing" in a given environment. We will need to be able to compute

a real value for any given (valid) chromosome.

The overall goal of the GA is to find member(s) of the population that optimize the fitness function.

- A process by which we can select which individuals will reproduce.
- A process by which chromosomes combine to create new chromosomes.

  This should also include some random mutation.

There are many different algorithms, and most differ on how one interprets the four previous items. Before we get too abstract, we'll work through a very simple example.

4.0.1 Example: Binary Strings

Suppose our problem is to maximize the number of 1's in a bit string of length 20. There is an easy solution to this- The optimal string would be a string of 1's, but let's see how our four pieces fit together in this problem:

1. The population: The population consists of strings of length 20, where each element of the string is a 0 or a 1. We also need the size of the population- In this case, we set it arbitrarily to 100.
2. The fitness function: The fitness function takes in a binary string of length 20, and outputs the number of 1's in the string.
3. A process by which individuals are selected for reproduction:

   In this example, we'll select individuals at random. Unfortunately, this is not specific enough, since there are many ways to make a "random" selection. In this case, if individual j has fitness value $f(j)$, then consider the set of probabilities:

   $$P_1 = \frac{f(1)}{\sum_{j=1}^{100} f(j)}, P_2 = \frac{f(2)}{\sum_{j=1}^{100} f(j)}, \ldots, P_{100} = \frac{f(100)}{\sum_{j=1}^{100} f(j)}$$

   In this example, these are indeed probabilities since they are non-negative and sum to 1. Therefore, we will define the selection process as: Select individual j with probability P_j. We should consider this problem separately, and write a separate function that will perform this function (and we will). This is actually a fairly standard way of selection.

   Here is the Matlab code we'll use. You might notice that it comes from code we used in the $n-$armed bandit problem, where we had to select machine a using probability P_a. Here, we'll call the function `RandChooseN`.

```
function Action=RandChooseN(P,N)
% function Action=RandChooseN(P,N)
%   Choose N numbers from 1 to length(P) using the
```

```
%   probabilities in P.  For example, if P=[0.1,0.9],
%   we choose "1" 10% of the time, and "2" 90% of
%   the time.   Selection is done WITH replacement,
%   so, for example, if N=3, we could return [2, 2, 2]

%Set up the bins
BinEdges=[0, cumsum(P(:)')];
Action=zeros(1,N);

for i=1:N
    x=rand;
    Counts=histc(x,BinEdges);
    Action(i)=find(Counts==1);
end
```

4. A process by which chromosomes combine to create new chromosomes:

   Chromosomes will be combined in two ways: Using crossover, then using a random mutation. We'll define the crossover process first.

   If we have two "parents", "ma" and "pa", each defined by a string of 20 characters, then in crossover we choose a crossover point that is an integer between 1 and 20. Let's call the crossover value x_p.

   The two offspring will then be two strings of length 20. The first uses the first x_p characters from "ma" and the remaining characters from the corresponding values in "pa".

   Similarly, the second offspring will use the initial x_p characters from "pa" and the remainder from "ma".

   Therefore, the offspring 1 would have its front part from "ma" and back part from "pa" (split at x_p). Offspring 2 is the opposite, its front part from "pa" and back part from "ma".

   The crossover point may be the same for all parents, or it can be randomly selected for each parent. In the code that follows, the crossover points were selected at random for each pair of parents.

   Furthermore, keeping with our analogy of evolution, we also include the possibility (usually a very small one) that mutations may occur, and they occur randomly.

   There are many other ways one could perform crossover and mutation- The only caution here is that one must ensure that the offspring are still valid chromosome strings.

Matlab Code for Example 1

Here is the Matlab code that implements the algorithms from the text. Save it in Matlab and run it several times and compare the outputs.

```
%% Genetic Algorithm Example
%
%  Problem:  Define the population as binary strings of length 20,
%  with the fitness function:  Sum of 1's.  Use a population of
%  100 with certain probabilities defined below to see how long
%  it takes to reach a string of all 1's.

%%  Setup the GA parameters

ff=inline('sum(x,2)'); % objective function

maxit=200;        % max number of iterations (for stopping)
maxcost=9999999; %Maximum allowable cost (for stopping)
popsize=100; % set population size (it is constant)
mutrate=0.001; % set mutation rate (a small probability)

lenx=20; % Length of the chromosome (20 here)

%% Create the initial population
pop=round(rand(popsize,lenx)); % random population of 1s
                               % and 0s (using default, 100x20)

% Initialize cost and other items to set up the main loop
cost=feval(ff,pop); % calculates population cost using ff

[cost,ind]=sort(cost,'descend'); % max element in first entry
pop=pop(ind,:); % sorts population with max cost first

probs=cost/sum(cost);  %Simple normalization for probabilities.

% We'll be tracking the following quantities for our plot:
maxc(1)=max(cost); % minc contains min of population
meanc(1)=mean(cost); % meanc contains mean of population


%% MAIN LOOP

iga=0;  % Initalize the variable used in the loop below.
        %  This will keep track of how many iterations we've
        %  used and will get us out of the loop at the max

while iga<maxit

    iga=iga+1; % increments generation counter

% Choose mates
```

```
    M=ceil(popsize/2); % number of matings
    ma=RandChooseN(probs,M); % mate #1
    pa=RandChooseN(probs,M); % mate #2
% ma and pa contain the *indices* of the chromosomes that will mate

% Select crossover values for each set of parents
    xp=randi([1,lenx],1,M);  % M integers from 1 to lenx
                             % In this code, crossover is different
                             %  for each set of parents.

    Temp=pop;  %Just temporary storage as we perform crossover

%Crossover: One offspring will be stored in the odd indices,
%           the other in the even indices.
    for k=1:M
      pop(2*k-1,:)=[Temp(ma(k),1:xp(k)) Temp(pa(k),xp(k)+1:20)];
      pop(2*k,:)=[Temp(pa(k),1:xp(k)) Temp(ma(k),xp(k)+1:20)];
    end

% Mutate the population
    nmut=ceil((popsize-1)*lenx*mutrate); % total number of mutations
    mrow=ceil(rand(1,nmut)*(popsize-1))+1; % row to mutate
    mcol=ceil(rand(1,nmut)*lenx); % column to mutate
    for ii=1:nmut
        pop(mrow(ii),mcol(ii))=abs(pop(mrow(ii),mcol(ii))-1); % toggles bits
    end % ii

%% The population is re-evaluated for cost
    cost=feval(ff,pop); % calculates population cost using ff
    [cost,ind]=sort(cost,'descend'); % max element in first entry
    pop=pop(ind,:); % sorts population with max cost first

    probs=cost/sum(cost);  %Re-set probabilities

% We keep track of some values for graphical output:
    maxc(iga+1)=max(cost);
    meanc(iga+1)=mean(cost);

%% Stopping criteria.  The double bar: || is an "or"
    if iga>maxit || cost(1)>maxcost
        break
    end

end %iga
```

```
%% Displays the output
day=clock;
disp(datestr(datenum(day(1),day(2),day(3),day(4),day(5),day(6)),0))
%disp(['optimized function is ' ff])
format short g
disp(['popsize = ' num2str(popsize) ' mutrate = ' num2str(mutrate)]);
disp(['#generations=' num2str(iga) ' best cost=' num2str(cost(1))]);
fprintf('best solution\n%s\n',mat2str(int8(pop(1,:))));
figure(1)
iters=0:length(maxc)-1;
plot(1:(iga+1),maxc,1:(iga+1),meanc);
xlabel('generation');ylabel('cost');
```

In this example, we have seen how to implement one GA on a population of binary strings. In the next section, we will consider changes if the string consists of real numbers.

4.0.2 Example Using Real Numbers

Consider the following problem: Find the minimum value of f over the domain $0 \leq x \leq 10$, $0 \leq y \leq 10$, and

$$f(x, y) = x \sin(4x) + 1.1y \sin(2y)$$

If one attempts to use "classical" techniques like gradient descent, then you rapidly get stuck in a local minimum (or a local max). The graph of f is shown in Figure 4.1.

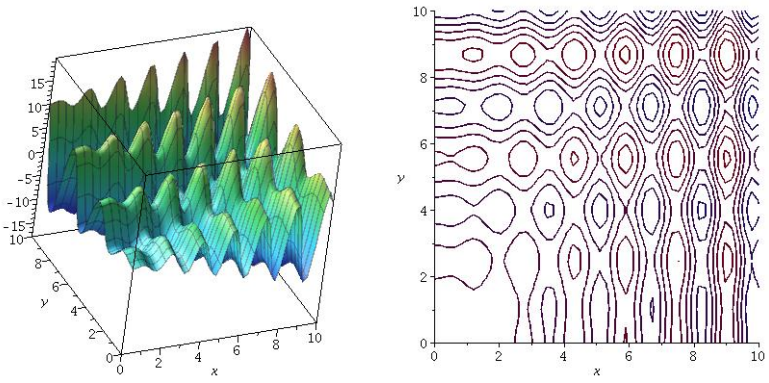

Figure 4.1: The surface and contour map for the function we will try to minimize, $f(x, y) = x \sin(4x) + 1.1y \sin(2y)$.

Now we'll work through the steps for defining the genetic algorithm:

1. Define the population. We will have 12 individuals (that number was randomly selected). Each chromosome will be an ordered pair of real numbers:
$$[x, y] \qquad 0 \leq x \leq 10, 0 \leq y \leq 10$$
The population will be initialized randomly.

2. Define the fitness function. The fitness function is given to us already:
$$f(x, y) = x \sin(4x) + 1.1y \sin(2y)$$

3. A process by which individuals are selected for reproduction:

   One interesting way of putting together the probability distribution is to use *rank order*. For example, suppose we have N things in order from "best" to "last". Then we choose item j with probability
$$\frac{N - (j-1)}{\sum_{i=1}^{N} i}$$
   with maximum probability $N/\sum i$ and minimum probability $1/\sum i$.

   For example, if we have 4 things, the probability distribution would be:
$$\frac{4}{10}, \quad \frac{3}{10}, \quad \frac{2}{10}, \quad \frac{1}{10}$$
   We'll implement this in the Matlab script.

4. Crossover and Mutation:

   This is where things get a little trickier. Here is one method that we implemented in Matlab below.

   - Select the crossover point at random (in this case, the x coordinate or the y coordinate).
   - The coordinate z_{ma}, z_{pa} selected for crossover will then be updated by selecting $0 < \beta < 1$:
$$z_{ma_{new}} = (1 - \beta) z_{ma} + \beta z_{pa}$$
$$z_{pa_{new}} = (1 - \beta) z_{pa} + \beta z_{ma}$$
   - The other coordinates will remain.
   - Since there are additional constraints on the coordinates, these will also be checked. That is, if the new x coordinate is greater than 10, it will be chopped to 10. Similarly, if a coordinate is less than zero, it will be chopped to zero.
   - If selected for mutation, a random number (uniform) will be selected between 0 and 10.

Matlab Script for the Optimization

```
%% Script File:  Optimization and GA

% Initialize the population:
Pop=10*rand(12,2);

% Stopping criteria
maxit=200;  %Max number of iterations
mincost=-99999999;  %Minimum cost

% Parameters:
popsize=12;
mutrate=0.05;  %Mutation rate
popKept=0.5;  %Fraction of the population to keep
keep=floor(popKept*popsize);  %How many individuals are kept
M=ceil((popsize-keep)/2); % number of matings; 2 mates create 2 offspring
crossprob=round(rand(maxit,1)); %0= x-coord, 1= y-coord
nmut=ceil((popsize-1)*2*mutrate);  %Number of mutations

% Fitness function:
ff='testfunction';  %The fitness function is in the file testfunction.m

% Probability distribution (won't change in this example)
probs=(keep:-1:1)/sum(1:keep);  %Probability is rank ordering

%% Initialize the population:

cost=feval(ff,Pop);  %Initial costs
[cost,idx]=sort(cost);  % Default sort is from small to large
Pop=Pop(idx,:);
minc(1)=min(cost);  %Minimum cost, for plotting later
meanc(1)=mean(cost);  %Mean cost for this population (for plotting later)


%% Main loop
iga=0;
while iga<maxit

    iga=iga+1;

    % Pair up and mate:
    ma=RandChooseN(probs,M);
    pa=RandChooseN(probs,M);

    % Set up crossover and mutation:
```

```
    idx2=keep+1:popsize;
    beta=rand;
    PopMa=Pop(ma,:);  PopPa=Pop(pa,:);
    if crossprob(iga)==0  %Crossover the x-coordinate
        for j=1:M
            Pop(idx2(2*j-1),1)=(1-beta)*PopMa(j,1)+beta*PopPa(j,1);
            Pop(idx2(2*j-1),2)=PopMa(j,2);
            Pop(idx2(2*j),1)=(1-beta)*PopPa(j,1)+beta*PopMa(j,1);
            Pop(idx2(2*j),2)=PopPa(j,2);
        end
    else  %Crossover the y-coordinate
        for j=1:M
            Pop(idx2(2*j-1),1)=PopMa(j,1);
            Pop(idx2(2*j-1),2)=(1-beta)*PopMa(j,2)+beta*PopPa(j,2);
            Pop(idx2(2*j),1)=PopPa(j,1);
            Pop(idx2(2*j),2)=(1-beta)*PopPa(j,2)+beta*PopMa(j,2);
        end
    end

    % Mutation

    mrow=sort(ceil(rand(1,nmut)*(popsize-1))+1);
    mcol=ceil(rand(1,nmut)*2);
    for ii=1:nmut
        Pop(mrow(ii),mcol(ii))=10*rand;
    end

    % New cost, set up for next iteration:
    cost=feval(ff,Pop);  %Initial costs
    [cost,idx]=sort(cost);  % Default sort is from small to large
    Pop=Pop(idx,:);
    minc(iga+1)=min(cost);  %Minimum cost, for plotting later
    meanc(iga+1)=mean(cost);  %Mean cost for this population (for plotting later)


    % Stopping criteria
    if iga>maxit || cost(1)<mincost
        break
    end

end % End of the while loop

%% Display the results
figure(1)
iters=0:length(minc)-1;
plot(iters,minc,iters,meanc,'-');
```

```
xlabel('generation');ylabel('cost');
```

4.0.3 Example: The Knapsack Problem

Suppose you want to go on a camping trip. You're planning on carrying no more than 20 kg of supplies. The problem is that you cannot possibly take all the supplies that you may want. An additional worry is that not every is kilogram is equal- For example, taking the tent should be much more important than taking a novel.

Therefore, you set up a scale on which you rate the relative importance of each item. Here is your list in Table 4.1.

Item	Value	Weight
bug repellent	12	2
camp stove	5	4
canteen (full)	10	7
clothes	11	5
dried food	50	3
first aid kit	15	3
flashlight	6	2
novel	4	2
rain gear	5	2
sleeping bag	25	3
tent	20	11
water filter	30	1

Table 4.1: Camping Supplies, with Weight and Value

Now we need to translate this problem into a GA (and there are many ways you might do it). Let's see if we can start.

One way to define the fitness function may be the following. The fitness value will be the sum of the values, if the weight is less than (or equal to) 20 kg. Otherwise, the fitness will take a value of -1. We will then try to maximize the fitness function.

One way to set things up is to make three strings of length 12 (for the number of items we could pack). One vector would store the values, one vector would store the weights, and the third vector could be binary, with 1 meaning "include this", and 0 meaning "do not include this".

Since we have 12 items total, we might define a chromosome as a vector with 1 (include this item), or 0 (do not include).

Exercises with the Knapsack Problem

We're going to go through the first example of a GA, the example with binary strings of length 20, and modify it to try to solve the knapsack problem.

Here are some suggestions for changes:

1. The objective function won't be so simple this time. Change the line:

   ```
   ff=inline(sum(x,2)); % objective function
   ```

   so that we can write the objective function as an M-file (like the second example).

   In the objective function, if X is a matrix that is 20×12 of zeros and ones, and `val` is a 1×12 vector of the "values" of each item, then what does the following command do? (You might look up the `repmat` command and what the `sum` command does if you use it like: `sum(A,2)`).

   ```
   sum(X.*repmat(val, numpop, 1), 2)
   ```

2. Some of the other parameters:

   - Change the maximum number of iterations to 250
   - We'll have a population size of 20, and we'll keep half. (The second example shows this).
   - The mutation rate will be about 10%

3. We'll keep the crossover technique and the way of selecting mates the same as for the binary strings.