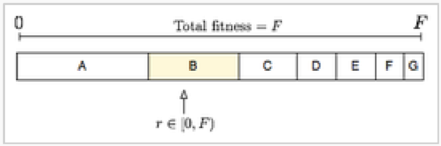
Fitness proportionate selection

From Wikipedia, the free encyclopedia



Example of the selection of a single individual

Fitness proportionate selection, also known as **roulette wheel selection**, is a genetic operator used in genetic algorithms for selecting potentially useful solutions for recombination.

In fitness proportionate selection, as in all selection methods, the fitness function assigns a fitness to possible solutions or chromosomes. This fitness level is used to associate a probability of selection with each individual chromosome. If f_i is the fitness of individual i in the population, its probability of being selected is $p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$, where N is the number of individuals in the population.

This could be imagined similar to a Roulette wheel in a casino. Usually a proportion of the wheel is assigned to each of the possible selections based on their fitness value. This could be achieved by dividing the fitness of a selection by the total fitness of all the selections, thereby normalizing them to 1. Then a random selection is made similar to how the roulette wheel is rotated.

While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. Contrast this with a less sophisticated selection algorithm, such as truncation selection, which will eliminate a fixed percentage of the weakest candidates. With fitness proportionate selection there is a chance some weaker solutions may survive the selection process; this is an advantage, as though a solution may be weak, it may include some component which could prove useful following the recombination process.

The analogy to a roulette wheel can be envisaged by imagining a roulette wheel in which each candidate solution represents a pocket on the wheel; the size of the pockets are proportionate to the probability of selection of the solution. Selecting N chromosomes from the population is equivalent to playing N games on the roulette wheel, as each candidate is drawn independently.

Other selection techniques, such as stochastic universal sampling[1] or tournament selection, are often used in practice. This is because they have less stochastic noise, or are fast, easy to implement and have a constant selection pressure [Blickle, 1996].

The naive implementation is carried out by first generating the cumulative probability distribution (CDF) over the list of individuals using a probability proportional to the fitness of the individual. A uniform random number from the range [0,1) is chosen and the inverse of the CDF for that number gives an individual. This corresponds to the roulette ball falling in the bin of an individual with a probability proportional to its width. The "bin" corresponding to the inverse of the uniform random number can be found most quickly by using a binary search over the elements of the CDF. It takes in the O(log n) time to choose an individual. A faster alternative that generates individuals in O(1) time will be to use the alias method.

Recently, a very simple O(1) algorithm was introduced that is based on "stochastic acceptance".[2] The algorithm randomly selects an individual (say i) and accepts the selection with probability f_i/f_M, where f_M is the maximum fitness in the population. Certain analysis indicates that the stochastic acceptance version has a considerably better performance than versions based on linear or binary search, especially in applications where fitness values might change during the run.[3]

Contents [hide]

Pseudocode [edit]

For example, if you have a population with fitnesses [1, 2, 3, 4], then the sum is 10 (1 + 2 + 3 + 4). Therefore, you would want the probabilities or chances to be [1/10, 2/10, 3/10, 4/10] or [0.1, 0.2, 0.3, 0.4]. If you were to visually normalize this between 0.0 and 1.0, it would be grouped like below with [red = 1/10, green = 2/10, blue = 3/10, black = 4/10]:

```
0.1 ]

0.2 \
0.3 /

0.4 \
0.5 |
0.6 /

0.7 \
0.8 |
0.9 |
1.0 /
```

Using the above example numbers, this is how to determine the probabilities:

```
sum_of_fitness = 10
previous_probability = 0.0

[1] = previous_probability + (fitness / sum_of_fitness) = 0.0 + (1 / 10) = 0.1
previous_probability = 0.1

[2] = previous_probability + (fitness / sum_of_fitness) = 0.1 + (2 / 10) = 0.3
previous_probability = 0.3

[3] = previous_probability + (fitness / sum_of_fitness) = 0.3 + (3 / 10) = 0.6
previous_probability = 0.6

[4] = previous_probability + (fitness / sum_of_fitness) = 0.6 + (4 / 10) = 1.0
```

The last index should always be 1.0 or close to it. Then this is how to randomly select an individual:

```
random_number # Between 0.0 and 1.0

if random_number < 0.1
  select
else if random_number < 0.3 # 0.3 - 0.1 = 0.2 probability
  select
else if random_number < 0.6 # 0.6 - 0.3 = 0.3 probability
  select
else if random_number < 1.0 # 1.0 - 0.6 = 0.4 probability
  select
end
```

Coding Examples [edit]

Java - linear O(n) version [edit]

```
// Returns the selected index based on the weights(probabilities)
int rouletteSelect(double[] weight) {
 // calculate the total weight
 double weight_sum = 0;
 for(int i=0; i<weight.length; i++) {
  weight_sum += weight[i];
 }
 // get a random value
 double value = randUniformPositive() * weight_sum;
 // locate the random value based on the weights
```

```
 for(int i=0; i<weight.length; i++) {
  value -= weight[i];
  if(value <= 0) return i;
 }
 // only when rounding errors occur
 return weight.length - 1;
}

// Returns a uniformly distributed double value between 0.0 and 1.0
double randUniformPositive() {
 // easiest implementation
 return new Random().nextDouble();
}
```

Java - stochastic acceptance O(1) version [edit]

```
public class roulette {
 /* program n_select=1000 times selects one of n=4 elements with weights weight[i].
  * Selections are summed up in counter[i]. For the weights as given in the example
  * below one expects that elements 0,1,2 and 3 will be selected (on average)
  * 200, 150, 600 and 50 times, respectively.  In good agreement with exemplary run.
  */
public static void main(String [] args) {
 int n=4;
 double [] weight = new double [n];
 weight[0]=0.4;
 weight[1]=0.3;
 weight[2]=1.2;
 weight[3]=0.1;
 double max_weight=1.2;
 int  [] counter = new int[n];
 int n_select=1000;
 int index=0;
 boolean notaccepted;
 for (int i=0; i<n_select; i++){
  notaccepted=true;
  while (notaccepted){
   index= (int)(n*Math.random());
   if(Math.random()<weight[index]/max_weight) {notaccepted=false;}
  }
  counter[index]++;
 }
 for (int i=0; i<n; i++){
  System.out.println("counter["+i+"]="+counter[i]);
 }
}
/* The program uses stochastic acceptance instead of linear (or binary) search.
 * More on http://arxiv.org/abs/1109.3627
 */
}
# Exemplary output:
# counter[0]=216
# counter[1]=135
# counter[2]=595
# counter[3]=54
```

Ruby - linear O(n) search [edit]

```
# Normalizes an array that potentially contains negative numbers by shifting
#   all of them up to be positive (0 is left alone).
#
# +pop_fit+ array of each individual's fitness in a population to normalize
def norm_pop(pop_fit)
  least_fit = pop_fit.min.abs + 1 # Absolute value so can shift up
                                  # +1 so that it doesn't become 0

  pop_fit.map! do |f|
    (f != 0) ? (f + least_fit) : f
```

```ruby
  end

  return pop_fit
end

# Returns an array of each individual's probability between 0.0 and 1.0 fitted
#   onto an imaginary roulette wheel (or pie).
#
# This will NOT work for negative fitness numbers, as a negative piece of a pie
#   (i.e., roulette wheel) does not make sense.  Therefore, if you have negative
#   numbers, you will have to normalize the population first before using this.
#
# +pop_fit+ array of each individual's fitness in the population
# +is_high_fit+ true if high fitness is best or false if low fitness is best
def get_probs(pop_fit,is_high_fit=true)
  fit_sum  = 0.0 # Sum of each individual's fitness in the population
  prob_sum = 0.0 # You can think of this in 2 ways; either...
                 # 1) Current sum of each individual's probability in the
                 #    population
                 # or...
                 # 2) Last (most recently processed) individual's probability
                 #    in the population
  probs    = []
  best_fit = nil # Only used if is_high_fit is false

  # Get fitness sum and best fitness
  pop_fit.each do |f|
    fit_sum += f

    if best_fit == nil or f > best_fit
      best_fit = f
    end
  end

  puts "Best fitness: #{best_fit}"
  puts "Fitness sum:  #{fit_sum}"

  best_fit += 1 # So that we don't get best_fit-best_fit=0

  # Get probabilities
  pop_fit.each_index do |i|
    f = pop_fit[i]

    if is_high_fit
      probs[i] = prob_sum + (f / fit_sum)
    else
      probs[i] = (f != 0) ? (prob_sum + ((best_fit - f) / fit_sum)) : 0.0
    end

    prob_sum = probs[i]
  end

  probs[probs.size - 1] = 1.0 # Ensure that the last individual is 1.0 due to
                              #   decimal problems in computers (can be 0.99...)
  return probs
end

# Selects and returns a random index using an array of probabilities that were
#   created to mirror a roulette wheel type of selection.
#
# +probs+ array of probabilities between 0.0 and 1.0 that total to 1.0
def roulette_select(probs)
  r = rand # Random number between 0.0 and 1.0

  probs.each_index do |i|
    if r < probs[i]
      return i
    end
  end

  return probs.size - 1 # This shouldn't happen
end
```

```ruby
pop_fit = [1,2,3,4]
pop_sum = Float(pop_fit.inject {|p,f| p + f})
probs = get_probs(pop_fit,true)

# These should all have the exact same output
puts probs.inspect
puts get_probs([4,3,2,1],false).inspect
puts get_probs(norm_pop([-4,-3,-2,-1]),true).inspect
puts get_probs(norm_pop([-1,-2,-3,-4]),false).inspect
puts

# Check the math
prev_prob = 0.0

puts "Math check:"
for i in 0..pop_fit.size-1
  puts "%.4f|%.4f|%.4f" % [probs[i],probs[i] - prev_prob,pop_fit[i] / pop_sum]
  prev_prob = probs[i]
end
puts

# Observe some random selections
observed_probs = Array.new(pop_fit.size,0)
observed_count = 1000

for i in 1..observed_count
  observed_probs[roulette_select(probs)] += 1
end

puts "Observed:"
observed_probs.each_index do |i|
  prob = observed_probs[i] / Float(observed_count)
  puts "#{i}: #{prob}"
end

# Example output:
#
# Best fitness: 4
# Fitness sum:  10.0
# [0.1, 0.30000000000000004, 0.6000000000000001, 1.0]
# Best fitness: 4
# Fitness sum:  10.0
# [0.1, 0.30000000000000004, 0.6000000000000001, 1.0]
# Best fitness: 4
# Fitness sum:  10.0
# [0.1, 0.30000000000000004, 0.6000000000000001, 1.0]
# Best fitness: 4
# Fitness sum:  10.0
# [0.1, 0.30000000000000004, 0.6000000000000001, 1.0]
#
# Math check:
# 0.1000|0.1000|0.1000
# 0.3000|0.2000|0.2000
# 0.6000|0.3000|0.3000
# 1.0000|0.4000|0.4000
#
# Observed:
# 0: 0.108
# 1: 0.191
# 2: 0.296
# 3: 0.405
```

See also [edit]

- Stochastic universal sampling
- Tournament selection
- Reward-based selection

External links [edit]

- C implementation (.tar.gz; see selector.cxx) WBL
- Example on Roulette wheel selection
- An outline of implementation of the O(1) version

References [edit]

1. ^ Bäck, Thomas, *Evolutionary Algorithms in Theory and Practice* (1996), p. 120, Oxford Univ. Press
2. ^ A. Lipowski, Roulette-wheel selection via stochastic acceptance (arXiv:1109.3627)[1]
3. ^ Fast Proportional Selection

Categories: Genetic algorithms