Genetic Algorithm

Due: Friday, March 29 @ 11:59pm

Updates and Pitfalls

-

Objective

Gain experience with functional programming techniques (Passing and returning functions)

Background

Reading:

- https://en.wikipedia.org/wiki/Genetic_algorithm
- https://www.tutorialspoint.com/genetic_algorithms/index.htm

Genetic algorithms provide probabilistic solutions to optimization problems. These algorithms can be thought of as an advanced “guess and check” technique that eventually arrives at an output that is close to the actual solution without having to know how to compute the solution directly.

Description

You will write a generic genetic algorithm that can find solutions to optimization problems. This algorithm will be written is such a way that it can be reused for any application that fits a genetic algorithm solution.

For the testing objectives your tasks are to apply your genetic algorithm to linear and polynomial regression. The goal for each of these is to find an equation that best fits a set of points. While these regression can be computed directly with an understanding of linear algebra, we will use these as tests for the genetic algorithm. One advantage of the genetic algorithm is that we can solve these regression problems without an understanding of linear algebra, which is a class you probably have not yet taken.

For this genetic algorithm we will use Doubles that range from 0.0 - 1.0 as the genes and provide a way to convert these genes into an object in the solution space. The algorithm also needs a fitness function that will measure how well a solution performs. The fitness function will return a Double in the range 0.0 - 1.0 with 1.0 being a prefect solution.

Project Structure

1. Create a new project in IntelliJ
2. Pull the Scala Examples repo and copy the regression package into the src folder in your new project
3. In your src folder add a package names tests where you'll add your test suites (The tests package is not a subpackage like it was in calculator)

Primary Objective (35/20 points)

Implement the generic genetic algorithm.

In the GeneticAlgorithm object write a method named geneticAlgorithm with:

- Takes a type parameter T
  - This is the type of the solution space that you are trying to find. For example, when computing linear regression this type will be Line
- As the first parameter, takes a fitness function of type T to Double
  - This function takes a potential solution and computes its fitness from 0.0 - 1.0 with 1.0 being a perfect solution
- As the second parameter, takes a function that takes a List[Gene] and returns a T
  - This function will provided a way to map a list of genes into an object in the solution space. This will involve mapping genes (Doubles from 0.0 - 1.0) into variable in the range needed in the solution space. For example, a line has a slope and y-intercept that can range from -Infinity to Infinity. For lines this function would take 2 genes and map them onto this range.
- As the third parameter, takes a List[Gene] as a sample gene for the solution space
  - This is used to let the algorithm know how many genes are needed for this application. In the line example this would be a list containing any 2 Genes.
- Returns an object of type T which is the most optimal solution found

Your algorithm will be tested with an unknown application to ensure you implemented the algorithm generically. Your algorithm will be called 200 times during primary objective testing to test the efficiency of your implementation. Be sure your algorithm is efficient enough to run this many times. Grading will timeout after 2 minutes.

Testing Objectives (30 points)

Testing Objective 1

In the GeneticAlgorithm object write a method named linearRegression that takes a list of points as a parameter and returns a line that best fits the points (the output of your genetic algorithm). Use the provided Point and Line classes. More specifically, return a line which minimizes the sum of the y distances of the points to the line.

To implement linearRegression you will need a way to generate the fitness function based on the provided set of points. The fitness function must take only an object from the solution space (A Line is this case). To generate the fitness function you should write a method that takes a list of points and returns a fitness function based on those points. This fitness function should measure sum of the distances of each point from the line in terms of their y values. For example if the points are (0,2) and (3,1) with a line y = x the total distance is 3.0. Do not use the sum of the squared distances.

You will also need to write a function/method that takes a list of genes and converts it into a line. For this function/method you should expect the input to have 2 genes (one for slope and one for y-intercept) and convert these genes into an object of type Line.

For the third argument to the genetic algorithm you will provide a sample gene. This can be any list containing 2 genes. This lets your algorithm know that your solution space will expect 2 genes.

In the tests package write a test suite named TestLinearRegression that tests this functionality. The genetic algorithm will not find the exact solutions so be sure to allow a fair amount of tolerance in your testing. The nearly correct solution will get close to the expected answer so don't allow too much tolerance. Fine tuning this value and ensuring your algorithm is getting close to the optimal solution will be important for this testing.

Hint: Since there is randomness in the algorithm it is a good idea to run your tests multiple times. The incorrect solutions will get lucky and be correct with a non-zero probability.

Hint: You will need to convert Doubles in the range 0.0 - 1.0 <---> -Infinity - Infinity in these functions. One way to accomplish this is by using tan and atan. For example, tan((geneValue - 0.5) * PI) converts from genes to -Infinity - Infinity. Keep in mind that this doesn't work very well for very large values in the solution space. We could address, but since it has little to do with functional programming you can simply avoid testing with large values. For primary object testing the solution space values will be in the range -100 - 100.

Testing Objective 2

In the GeneticAlgorithm object write a method named polynomialRegression that takes a list of points and an int representing the degree of the polynomial as parameters and returns a polynomial that best fits the points (the output of your genetic algorithm). Use the provided Point and Polynomial classes.

This regression is very similar to linear regression except the caller specifies the degree of the polynomial to used in the regression instead of always using a line (degree 1). Recall that you will have degree + 1 coefficients to represent the polynomial and should use degree + 1 genes.

In the tests package write a test suite named TestPolynomialRegression that tests this functionality. Since there is a large amount of variance when high degree polynomials are used, which makes testing difficult, only use degree 2 polynomials for testing.

Recommended Algorithm

You are free to implement your genetic algorithm any way you'd like as long as it follows the structure of the assignment. However, since this is not a course on randomized algorithms a recommended implementation is provided here.

- Start with 20 random "animals" with purely random genes
    - Use the provided sample gene (3rd parameter) to know how many genes to create for each animal
    - Use Math.random() to generate random Doubles in the range 0.0 - 1.0
- [Start Generation] Use the function from the second parameter to convert each list of genes into an object in the solution space
- Use the fitness function to find the fitness of each solution
- Setup the next generation by:
    - Adding the most fit animal to the next generation
    - Adding 2 mutations of the most fit and 1 mutation of the 2nd most fit animal to the next generation
        - Mutate the genes of a solution by adding a 1% variation to each gene. Use Math.random() to obtain a value from 0.0 - 1.0 then scale this to get a 1% change, then add that change to the existing gene
        - Be sure to not mutate into a gene > 1.0 or < 0.0
    - Add children using all combinations of the 4 most fit solutions as parents (6 total) to the next generation
        - Combine two solutions by taking the average of each of their genes
    - Add 10 random animals to the next generation
- Repeat for 10000 generations