CSC 395 Homework 3 Spring 2024

Due: Wednesday, March 13

In this homework assignment, you will be creating code to fit polynomials to data using a genetic algorithm. Specifically, you will be given a data set in the form of a list of x values and corresponding y values. Your task will be to fit an nth order polynomial to this data that minimizes the average squared error by determining the value of the coefficients c_i, where the polynomial is

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n.$$

Note that the value of n will be given for each scenario we consider.

You will be submitting your code in the file `hw3.py` to the course gradescope. Please do not alter the provided `Organism.py` file.

1 Organism Class

You have been provided with a class `Organism` to use for your genetic algorithm, in the file `Organism.py`. You should familiarize yourself with this class before beginning to write any code.

This class has 5 attributes:

- `bits`: the array of bits (0 or 1) representing the coefficients of the fitted polynomial for this particular organism.
- `floats`: a list of the corresponding floating point value of the coefficients.
- `fitness`: this organism's fitness, to be computed externally.
- `normFit`: the normalized fitness of the organism.
- `accFit`: the accumulated normalized fitness value used during selection.

The `Organism` constructor takes in a required input integer `numCoeffs`, which is the number of coefficients in the polynomial. For example, a cubic polynomial has 4 coefficients. The constructor also has an optional input `bits`, allowing the user to specify the bit list for the created organism (which will be useful when performing the crossover and mutation operations). If no input `bits` is given, a random list of bits will be created. Initially the fitness values are all set to 0, as they will be updated externally.

This `Organism` class also has a `__repr__` function to print the fitted solution in a more readable format, and a set of overloaded comparison operators that will allow for sorting Organisms by their fitness values. The `isClone` method can be used to check if two organisms represent the same solution (by checking if their bit arrays are the same).

The file also includes the `getFloats` function, which will convert a bit list into the corresponding list of floating point coefficients. This function is used by the constructor and should not be used elsewhere. Note that when a new organism has been created, it has the correct floating point coefficients.

2 Provided Code in `hw3.py`

You have also been provided with the file `hw3.py`, which will contain your code for the genetic algorithm. You have been provided with several fully written functions as well as placeholders for the functions you will be expected to create. *You should pay careful attention to the expected inputs and outputs of the different functions.*

First note the included import statement: `import Organism as Org`.

In this `hw3.py` file, you have been provided with three fully written functions, and the entire `main` block for the different scenarios we will consider (these scenarios will be discussed later).

- The `calcFit` function will calculate the fit of a given organism. Specifically, it will take in an `Organism` object and two lists: `xVals` and `yVals`. These two lists are the data we are fitting. Note that we are using lists to avoid strange numpy "warnings" that get printed but are not actually raised, making them difficult to catch and process as desired.

  We are trying to fit a polynomial so that we minimize the average squared error. Specifically, if y_i are the values of the data at the sample locations x_i (i.e., y_i are stored in `yVals` and x_i are stored in `xVals`), and if Y_i are the value of the fitted polynomial evaluated at the x_i, then the error is

  $$error = \frac{1}{N}\sqrt{\sum_{i=0}^{N-1}(y_i - Y_i)^2},$$

  where N is the number of data points, i.e., `len(xVals)`. The function considers each x value, and for each, computes the fitted value Y_i by evaluating the polynomial, and keeps a running sum of the squared error. Note that it is possible that we might be squaring massively large values (after all, our initial population will be random), which can cause an `OverflowError`. In this case, we will set the error to be `math.inf`.

  Finally, the returned fitness value will be the reciprocal of the error so that maximizing fitness will minimize the error. Note that we end by checking if the fitness value `isnan`, i.e., is 'not a number' (which can occur sometimes when performing operations with `math.inf`). If this is the case, we set the fitness value to 0.

- The `initPop` function will create the initial population. It takes 2 inputs, the size of the population as the integer `size`, and the number of coefficients for the polynomial fit as the integer `numCoeffs`. It then generates `size-4` random organisms to make up the population. The remaining 4 organisms are created specially to help sample a useful portion of the domain. They are:

  - The organism with all bits set to 0.
  - The organism with all bits set to 1.
  - The organism with all coefficients set to 1.
  - The organism with all coefficients set to -1.

These last two organisms are critical for the performance of our fits, since they help the algorithm by seeding it with both positive and negative $O(1)$ coefficients. The randomly generated organisms will tend to have either very large or very small coefficients because the bits representing the exponent are randomly chosen. In these two organisms, those bits are set to correctly represent the number 2^0 and guarantee that the algorithm is seeded with some more reasonable valued coefficients.

- The `runScenario` function takes in a string label for a scenario and the GA parameters. It then runs the GA code, plots the highest fitness score vs generation number, and returns a list of the best individuals found during the algorithm.
- **At the top of the provided `main` block, you can activate/deactivate any of the four scenarios by setting the values of `scenA`, `scenB`, `scenC`, and `scenD` to `True`/`False`.**

3 Required Code

You will be required to write the code for 6 functions.

- The `crossover` function will perform the crossover operation for the genetic algorithm. For this function you will be performing single-point crossover. So you will randomly choose a single index k, which you will use to split the parents' genomes when creating the two offspring.
- The `mutation` function will perform the mutation operation for the genetic algorithm. For this function, you should consider each bit in the `genome`, and with probability given by `mutRate`, flip the bit under consideration. When you have completed this process, return the mutated `genome`. It may be helpful to note that $1 - x$ is 1 if $x = 0$, and is 0 if $x = 1$, i.e., this simple arithmetic will perform a bit flip.
- The `selection` function will select a single organism for mating from the input population (which must be a list of `Organism` objects sorted in descending order by fitness, with all accumulated fitness values already computed). You should find the first organism with an `accFit` attribute greater than some random number you chose in the range $(0, 1)$. Return that organism. If you reach the end of the population list without finding such an organism, you should return the last organism in the population, i.e., `pop[-1]`.
- The `accPop` function will calculate the fitness, normalized fitness, and accumulated fitness values for the entire population, and sort the population in descending order of fitness. It takes as inputs the list of `Organism` objects `pop`, and the **lists** of the x and y values of the provided data, `xVals` and `yVals`, to be used for calculating the fitness (see the `calcFit` fitness function's description in the previous section).

  You will need to compute the fitness of each organism, sort the population in descending order of fitness (`pop.sort(reverse=True)`), normalize the fitness values according to the total fitness of the whole population, and compute each organism's accumulated normalized fitness.

- The `nextGeneration` will create the new generation given the old one. It will take as inputs the sorted list of `Organism` objects with accumulated fitnesses computed, `pop`; the integer number of coefficients in the polynomials `numCoeffs`; the floating point valued mutation rate `mutRate`; and the integer number of 'elite' individuals `eliteNum`.

  We start with two observations. First, note that we will be keeping `eliteNum` number of individuals from the old population, so we only need to create `len(pop)-eliteNum` number of new children. Second, note that the crossover operation creates 2 children at once.

  This means that we can start with an empty list `newPop` and append pairs of children produced as the result of `(len(pop)-eliteNum)//2` matings. For each mating:

  – Select two parents using two calls to the `selection` function.
  – Create the genomes of the two children using the `crossover` function.
  – Mutate each child's genome.

  Once all of the new children have been created, you need to append the best `eliteNum` individuals from the old `pop`.

- The final function to implement is the `GA` function that performs the genetic algorithm. This function must take in the following inputs in order:

  – `k`: the number of generations to create.
  – `size`: the size of the population.
  – `numCoeffs`: the number of coefficients in the polynomials.
  – `mutRate`: the mutation rate.
  – `xVals`: the x values of the given data.
  – `yVals`: the y values of the given data.
  – `eliteNum`: the number of elite individuals to keep per generation.
  – `bestN`: the number of best individuals seen so far to track over time.

  This function will perform the genetic algorithm and return 2 values. The output `best` should be a sorted list of `Organism` objects corresponding to the `bestN` number of *unique* individuals with the highest fitness observed over the course of the algorithm. The output `fit` will be a list of the highest fitness seen so far for each generation (i.e., these values can only increase).

  Note that as you track the best individuals seen so far, you should ensure that you don't have clones in your list (i.e., only add a new individual to the list if it is better than at least one individual already on the list, and is not a clone of any individual already on the list).

  You should now be able to fully implement the genetic algorithm using the rest of the code you have completed.

4 Four Scenarios in the Main Function

You have been provided with a fully written `main` function and a `runScenario` function that contains 4 scenarios along with the code to run the algorithm and print/plot the results. I fully trust that you could write the code to define the parameters, call the `GA` function, and print/plot the results. So this code is provided for the sake of saving you time to focus on the more complex parts of this algorithm. The provided scenarios correspond to fitting polynomials to the functions:

A. $y = 1$

B. $y = 5$

C. $y = x^2 - 1$

D. $y = \cos x$

In each scenario, we will consider several hundred generations with a population size of 1000, with the elite portion being the top 5% of the population, i.e., 50 individuals.

The mutation rate has been set rather high at 10%. The reason for this change is that most of the bits correspond to either very tiny changes in the number (i.e., in rather insignificant digits), or to very massive changes in the number (i.e., in the exponent). When mutated, the former will result in almost no noticeable change, while the latter will likely result in a very low fitness individual. A higher mutation rate will make it more likely that we see the intermediate changes that will give us better fits, while our selection process will remove the damaging large changes. **You are encouraged to explore changing the different parameters to see how they affect the results.**

For the first 3 scenarios, we will be using cubic polynomials to fit the data. Note that these inputs can be represented exactly by cubic polynomials. So these will provide a test to ensure that the algorithm is performing correctly. We should, in these cases, see something approaching the exact solution.

Note that it is possible to get unlucky, so it will probably be necessary to run each scenario multiple times until you see the desired performance.

For the scenario D, we will use a quartic polynomial to fit the function $y = \cos x$ on the domain $x \in [-5, 5]$. Now clearly this is not possible to do exactly. However, once we are sure that our algorithm is performing correctly on the other test scenarios, we can still attempt to perform this fit. We can even make some arguments for what solutions we would expect to see: $\cos x$ is an even function, so the resulting fit should prioritize the even powers of x.

5 Submission

You should submit your completed `hw3.py` file to the course gradescope. Please do not make any alterations to the provided `Organism.py` file.