In this assignment, your task is to use genetic programming (GP) to implement symbolic regression. Begin by reading the following tutorial by John Koza (if you haven't already), one of the world's leading experts on applying genetic programming techniques to solve challenging optimization problems.

`http://www.genetic-programming.com/gpquadraticexample.html`

About the Supplementary Files

You are provided a zip archive that contains three files: `Generator1.jar`, `Generator2.jar`, and `data.txt`.

- `Generator1.jar`: This is a compiled Java executable file. Given an x value as input, it outputs the value of a mystery function f evaluated at that point. Your task is to use symbolic regression to determine what the function f is. Note that f is a function that can expressed solely by composing the arithmetic operators $+, -, \times, \div$, positive integer powers of x, and integer constants. Here's an example of how to use this generator to print out the value of $f(0.15)$:
  `java -jar Generator1.jar 0.15`

- `data.txt`: This text file contains $100,000$ data points collected from an experiment, with one data point per line. The first three columns contain the measured values of the variables x_1, x_2 and x_3. The last column is the measured value of $y = f(x_1, x_2, x_3)$. Your task again is to determine the function f using symbolic regression. You may assume that f can be expressed solely by composing $+, -, \times, \div$, positive integer powers of x_1, x_2 and x_3, and real-valued constants.

- `Generator2.java`: Like `Generator1.jar`, this too is a generator that evaluates a mystery function f at a given point x. The function that is evaluated by `Generator2`, however, is quite a bit more complicated than that used by `Generator1`. When performing symbolic regression, in addition to $+, -, \times, \div$, integer powers of x, and integer constants, you will also need to include other analytic functions in your set of building blocks: for example, $e^x, \sin x, \log x$ etc. Use this generator in the same way as `Generator1`:
  `java -jar Generator2.jar 1.85`

Important Note: For this assignment, performing symbolic regression on the data generated by `Generator1.jar` and the data contained in `data.txt` is *required*. Performing regression on `Generator2.jar` is an optional, extra-credit opportunity.

On Overfitting

A common problem that one often encounters when performing regression is *overfitting*. The first five minutes of the following video offers a nice explanation of the problem.

`http://youtu.be/Tl5JB5EuU5o`

How might this affect your results? Here's a possible scenario: suppose you decide to carry out 10 independent runs of GP on `data.txt` to determine the form the function. In each of these runs, you use all $100,000$ data points when evaluating the fitness of your individuals (i.e., the squared error between your models' predictions and the true function value). After 10 independent runs that each started with a different random population of individuals, you are left with 10 different candidate models $f_1, f_2, \ldots f_{10}$, each of which was the best performing individual from the corresponding run. Which one of these is the overall best? An obvious approach would be to choose the f_i that has the lowest squared error among these 10 candidates. But now, you will have overfit to your dataset — the f_i that has the lowest error on these $100,000$ points may not necessarily have the best *generalization error*, i.e., may not be the best performing model on unseen examples. A common way to deal with this problem is to partition your data into a "training set" and a "test set". For example, you could set aside $20,000$ data points from `data.txt` as your test set with the remainder forming the training set. Now, when running GP, you evaluate the fitness of your individuals *only* on the training set. As before, let's say you now generate 10 models $f_1, f_2, \ldots f_{10}$ from 10 independent runs. You can now pick the best overall performer among these models by looking at how well they perform on the *test set*, i.e., data points that these models were not exposed to during the training process0. This will help mitigate overfitting.

There is another popular technique for minimizing overfitting in GP: you can bake into your fitness function a penalty for overly complex hypotheses. In the machine learning community, this trick is referred to as *regularization* — it encourages the algorithm to try to fit the data using simple hypotheses first, before trying more complex ones. For example, in your fitness function, you could impose a penalty on trees that are too deep. Or too large. Or something else you choose to design, or adapt from the existing literature. For best results, you will want to incorporate both mechanisms, i.e., use a train-test split for evaluating individuals and a "complexity" penalty in your fitness function.

The Write-Up

Remember the overarching writing rule for this course: *you need to be sufficiently precise with your writing and include enough details that a competent reader could reproduce your results.* Here are some specific things to address in your write-up, in no particular order. This is *not* meant to be an exhaustive list.

- What was the fitness function that you used? Was there a complexity/regularization term? How was the latter designed?
- How did you generate your initial "seed" population of individuals?
- What were your various parameter settings? For example: the population size, the number of generations of evolution, the mutation rate, the number of independent evolutionary runs etc. Were these choices made arbitrarily or did you do some systematic testing before setting these values? Where appropriate, describe your parameter selection process.
- Was loss of diversity a problem? Did you use any diversity preservation techniques?
- Describe your scheme for avoiding overfitting. What was the size of your train-test split?
- Include relevant data and/or charts in your experiments/results section. Don't forget to identify your best-performing models: what do you think is $f(x)$ for `Generator1.jar` and $f(x_1, x_2, x_3)$ for `data.txt`?
- If you attempted the extra-credit portion, include relevant details about those experiments as well: what building blocks you used in constructing the individuals, what you believe $f(x)$ is, etc.

Program Design Advice: design a class for representing individuals (i.e., trees) in your population. The GP algorithm itself can then be written using a series of functions/static methods.

Deliverables

Your primary deliverable for this assignment is a write-up composed in LaTeX, formatted using the AAAI template. The write-up should be no more than six pages in length, including references. Please submit a hardcopy of your write-up to my office and upload a zip archive of your code to Moodle by the due date.