In this assignment, your task is to use genetic programming (GP) to implement symbolic regression. Begin by reading the following tutorial by John Koza, 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: `dataset1.csv`, `dataset2.csv`, and `dataset3.csv`. The comma-separated values (CSV) format is a popular plain-text data format — you should be able to examine these files using Excel, or just a regular text editor. You can read the data in these files into your programs just as you would any other text file[1].

- `dataset1.csv`: This file contains $25,000$ data points, one point per line. Each line contains an x value and the corresponding value of a mystery function f that has been evaluated at that x value. 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 $+, -, \times, \div$, and integer constants.

- `dataset2.csv`: This file contains $25,000$ 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 $+, -, \times, \div$ and **real-valued** constants.

- `dataset3.csv`: This file is similar to `dataset1.csv` in that it too provides $25,000$ data points corresponding to evaluations of a mystery function $f(x)$. This function, however, is quite a bit more complicated than the one that was used to generate `dataset1.csv`. When performing symbolic regression, in addition to $+, -, \times, \div$ 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.

For this assignment, performing symbolic regression on the data provided in `dataset1.csv` and `dataset2.csv` is *required*. Performing regression on `dataset3.csv` is an optional, extra-credit opportunity.

[1] Indeed, there are nice free libraries for reading and writing CSV files for both Python and Java, that may also prove to be useful. Don't waste your time writing your own parser!

On Overfitting

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

`https://youtu.be/u73PU6Qwl1I`

How might this affect your results? Here's a possible scenario: suppose you decide to carry out 10 independent runs of GP on `dataset1.csv` to determine the form of the function. In each of these runs, you use all $25,000$ 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 $25,000$ 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 $5,000$ data points from `dataset1.csv` 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 process. This will help mitigate overfitting. You can now report the performance of this best individual on the $5,000$ data points that were held out — since the GP process never got to see this data, the error on these points is an unbiased estimate of the model's true performance.

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 idea 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 preferably, 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? (**Aside**: lots of people have spent a lot of time designing bloat-combating techniques, so your best bet would be to do some reading first!)
- 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. It may be helpful to summarize them in a table. 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 `dataset1.csv` and $f(x_1, x_2, x_3)$ for `dataset2.csv`?
- 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 and manipulating individuals (i.e., trees) in your population. I *highly* recommend that your trees be designed to be immutable, i.e., so that they cannot be modified after the initial construction. Operations like mutation and crossover would return *new* tree objects, rather than modifying the objects that called them. Making classes immutable whenever possible is a good design principle. The behavior of immutable objects is easier to reason about and predict, and will likely protect you from dealing with difficult to trace memory bugs arising from reference aliasing. The GP algorithm itself can then be written using a series of functions/static methods.

Recommended Timetable

Here's a recommendation for how to budget your time over the next couple of weeks as you work on this assignment.

- **Feb. 16–20:** Read the problem description, write code for reading in your data, think about how to design your `Tree` class, write and debug your `Tree` code for representing and manipulating individuals, prepare for your code design check-in with Dr. Ramanujan.
- **Feb. 21–24:** Meet with Dr. Ramanujan, revisit your design choices and adjust/refactor your design (if needed) based on your conversations, continue implementing your `Tree` class, start thinking about the design of your GP algorithm.
- **Feb. 25–28:** Do background research on GP techniques (for example: techniques for selection, bloat control, etc.), implement and debug your GP algorithm, run preliminary experiments on `dataset1.csv`, write your *Introduction* and *Background* sections.
- **Mar. 1–3:** Continue debugging/tuning your GP algorithm, run experiments on `dataset1.csv` and `dataset2.csv`, write drafts of your *Experiments* and *Results* sections.
- **Mar. 4–12:** Spring break, woohoo!
- **Mar. 13–16:** Wrap-up any pending experiments, write the *Conclusions* section and the abstract, revise and proofread the entire report and submit it.