COMP704 week 12: Genetic Programming

29 April 2020

Introduction

In the past few lectures, we have looked at evolutionary algorithms and genetic programming. In this workshop, we will explore genetic programming as a problem solving technique, particularly looking at it as an alternative approach for the problems you have been looking at in your assignments.

Introducing DEAP

DEAP (Distributed Evolutionary Algorithms in Python) is a Python library implementing many common evolutionary algorithms and genetic programming techniques. Begin by looking at the following example from the DEAP documentation:

`https://deap.readthedocs.io/en/master/examples/gp_symbreg.html`

Download the code from the link at the bottom of the page, and get it running within PyCharm. You will probably need to install the deap package, which as usual can be done using pip or using PyCharm's built-in package manager.

Examining the output

The example outputs some statistics from the genetic programming algorithm, but does not output the solution itself. Try adding the following line to the `main()` function, just before the `return` statement:

```
94      print(hof[0])
```

This gets the fittest individual from the *hall of fame* (what we referred to as the *elite* in lectures) and prints it. Convince yourself that the output does match up with the expression $x^4 + x^3 + x^2 + x$ which it is trying to evolve.

Doing something more useful

This example is trying to find a mathematical expression in one variable. We can easily add more variables by changing a few lines:

```
37    pset = gp.PrimitiveSet("MAIN", 1)
```

The `1` here is the number of variables.

```
46    pset.renameArguments(ARG0='x')
```

If we have more than one variable, we can add arguments here to give them meaningful names e.g.:

```
46    pset.renameArguments(ARG0='x', ARG1='y', ARG2='z')
```

Now looking at the fitness function:

```
63    sqerrors = ((func(x) - x ** 4 - x ** 3 - x ** 2 -
          x) ** 2 for x in points)
```

Here `func` is the phenotype — the GP individual compiled into a Python function. If you have more than one variable, `func` will have more than one argument. Let's look at line 63 alongside line 67:

```
67    toolbox.register("evaluate", evalSymbReg, points=[
          x / 10. for x in range(-10, 10)])
```

So `points` is a list of numbers $[-1, -0.9, \ldots, 0.8, 0.9]$. These get passed into the `evalSymbReg` function, and line 63 loops through calculating

$$\big(\text{func}(x) - \big(x^4 + x^3 + x^2 + x\big)\big)^2$$

for each value. That is, the square of the difference between the individual's value and the target polynomial's value — averaging these squared differences gives the *mean squared error*.

Let's say instead that we have a table of data, represented as a list of lists. By passing the data table in as `points`, we could do something like

```
sqerrors = ((func(a, b, c, d) - out) ** 2
    for (a, b, c, d, out) in points)
```

(Obviously you should use more descriptive names than `a, b, c, d`!) Thus we can use GP to find a mathematical function to fit the data we pass in.

Experiment with one of your datasets from your assignment project, loading it in to this example and trying to find a mathematical expression to predict the output value based on the input values. You will probably need to add some more primitive node types to get reasonable results — look at the other GP examples in the DEAP documentation for ideas (the `if_then_else` primitive from the 3–8 multiplexer example is likely to be particularly useful). You may also want to play with the parameters to the GP algorithm, particularly the number of generations (which is set to 40 by default).

Using the results

If you manage to find a solution that has a good error rate, try implementing it into your game and seeing how well it plays. You *could* do this by examining the printed output of `print(hof[0])` and transcribing it to Python code by hand, but a better approach for you to explore is to use pickling to save `hof[0]` and load it into your game code. Good luck!