Background to study and datasets used

Background

This assignment was about cost estimation and genetic programming. Cost estimation is the problem of estimating how much a project would cost given a number of attributes about it and the same attributes and costs of a number of other projects. One approach is to find a relationship between the attributes and costs of projects that have already been done and use it to estimate the cost of a hypothetical project. In other words, cost estimation can be solved as a search problem where the thing that is being searched for is a function that takes attributes and returns a cost.

Genetic programming is the technique used to do this. Functions "are encoded as a set of genes that are then modified using an evolutionary algorithm"[1]. The encoding takes the form of a tree, Figure 1, where each node is an operator (like + or -) and each leaf is an operand (like an input to the function or a random constant). Encoding functions like this means a genetic algorithm can be used to find the function that most accurately maps the attributes of a project to its cost.

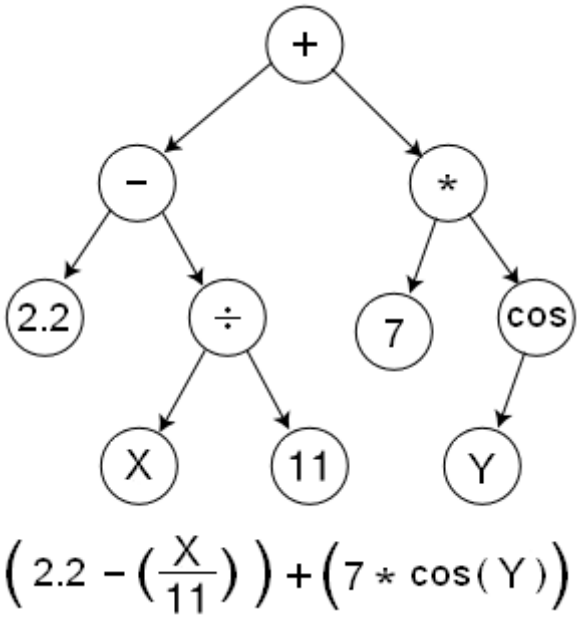

Figure 1: A function represented as a tree structure.[2]

I talked about how genetic algorithms work in my other assignments and won't waste your time repeating myself.

[1] https://en.wikipedia.org/wiki/Genetic_programming
[2] https://commons.wikimedia.org/wiki/File:Genetic_Program_Tree.png

Datasets

I used data from the PROMISE Software Engineering Repository[3] for this assignment. The repository itself was down but I downloaded seven datasets from TunedIT[4] and ten from MyPlace. The ones I ended up using were kemerer.arff, miyazaki94.arff and albrecht.arff, Table 1.

Table 1: Comparison of kemerer.arff, miyazaki94.arff and albrecht.arff.

kemerer.arff	Size: 15	**miyazaki94.arff**	Size: 48	**albrecht.arff**	Size: 24
Attribute	**Type**	**Attribute**	**Type**	**Attribute**	**Type**
ID	numeric	ID	{A1,…,T1}	Input	numeric
Language	numeric	KLOC	numeric	Output	numeric
Hardware	numeric	SCRN	numeric	Inquiry	numeric
Duration	numeric	FORM	numeric	File	numeric
KSLOC	numeric	FILE	numeric	FPAdj	numeric
AdjFP	numeric	ESCRN	numeric	RawFPcounts	numeric
RAWFP	numeric	EFORM	numeric	AdjFP	numeric
EffortMM	numeric	EFILE	numeric	Effort	numeric
		MM	numeric		

Attributes

I had to decide which attributes to include and which to ignore. I decided to ignore non-numeric attributes (of which there was just one, ID in miyazaki94.arff) because for attributes to be added, subtracted, multiplied and divided, they had to be numeric. (Other datasets like cocomo-sdr.arff and nasa93-dem.arff were dominated by non-numeric attributes and wouldn't have made sense to use.) The other question in my mind was whether to manually ignore attributes like ID that have no affect on cost. I think either answer would have been okay. Ultimately, I didn't ignore any numeric attributes and hoped the genetic algorithm would learn which attributes did and didn't matter.

[3] http://promise.site.uottawa.ca/SERepository/
[4] http://tunedit.org/repo/PROMISE/EffortPrediction

Implementation details

I wanted to continue experimenting with Python. Of the four frameworks suggested on MyPlace, Pyvolution and Pyevolve (as far as I could tell) don't have any genetic programming features, leaving deap and pySTEP. I flipped a coin and got deap. It was an "all in one" solution because it has both evolutionary algorithms and genetic programming. It wasn't easy to get my head around the tutorials and examples in the documentation. I was less confident than I would have liked and had less time than I would have liked because of other deadlines.

I started by copying the first example[5] and trying to modify it to evolve a function for kemerer.arff. I changed the number of inputs to 7 (the number of attributes in kemerer.arff excluding EffortMM (cost)), removed the cos and sin operators from the function set (because I didn't find them to be useful), manually renamed the arguments ID, Language, etc., wrote my own evaluation function (more on that later) and manually fed it 12 of the 15 projects from kemerer.arff (with the intention to test the evolved function on the other three, an 80/20 train/test split). I saved this as kemerer.py.

Three standard measures to assess how good a prediction is were covered in the lecture, MMRE (Mean Magnitude of Relative Error), Pred(n) and MAE (Mean Absolute Error). I used MMRE because the error had to be relative, not absolute like in MAE, to be able to compare different datasets. (Pred(n) is the proportion of estimates within n% of the actual and rewards being consistently fairly good and reliable.)

Part two was to go from everything being hard-coded for kemerer.arff to being able to use any dataset. I divided this into three main parts:

- Read attributes and data from dataset, ignoring non-numeric attributes
- Pass the number of arguments and rename them at runtime
- Rewrite the evaluation function

I wrote a read function that can read the three datasets I was going to use. I found out too late that it can't read some of the other datasets because they have attributes like …, [0,1], … and splitting this by ',' returns […, [0, 1], …]. I had no time to fix this. Sorry…

```
with open(file + '.arff') as f:
        read_data = f.readlines()
    for x in range(len(read_data)):
        if read_data[x][0] == '@':
            if read_data[x].split()[0] == '@attribute':
                if read_data[x].split()[2] == 'numeric':
                    attributes.append(read_data[x].split()[1])
                numerics.append(read_data[x].split()[2] == 'numeric')
```

[5] https://deap.readthedocs.io/en/master/examples/gp_symbreg.html

```
            elif read_data[x] == '@data\n':
                break
    x += 1
    for y in range(x, len(read_data)):
        row = read_data[y].strip().split(',')
        a = []
        for z in range(len(row)):
            if numerics[z]:
                a.append(float(row[z]))
        datas.append(a)
```

Passing the number of arguments was as easy as `len(attributes) - 1` (because every attribute except cost is an argument) but renaming them was less easy. The only way to rename arguments is to call `renameArguments()` which, contrary to its name, renames one argument at a time. For example, `pset.renameArguments(ARG0='ID')` renames the first argument ID. As much as I wanted to be able to call `renameArguments(attributes[:-1])`, I couldn't. The only way I could find was…

```
for x in range(len(attributes) - 1):
        exec('pset.renameArguments(ARG'+str(x)+'=\''+
                                    attributes[x] + '\')')
```

… which has "bad practice" written all over it. I converted the evaluation function from

```
def mmre(individual, points):
    func = toolbox.compile(expr=individual)
    mres = (abs(x[7] - func(x[0], x[1], x[2], x[3], x[4], x[5], x[6]))
            / x[7] for x in points                                    )
    return math.fsum(mres) / len(points),
```

to…

```
def mmre(individual, points):
        func = toolbox.compile(expr=individual)
        mres = (abs(x[len(x) - 1] - func(*x[0:len(x)-1]))
                / x[len(x) - 1] for x in points          )
        return math.fsum(mres) / len(points),
```

… making it flexible. To achieve the train/test split, I manually copied some of the lines from below @data to above it (3 lines in kemerer.arff, 5 lines in miyazaki94.arff and 5 lines in albrecht.arff). I changed the maximum tree height from 17 (what it was in the example) to 6 at the last minute to make the evolved functions a bit less monstrous and save time.

Presentation of results

To figure out how many generations to run the genetic algorithm for (ngen), I conducted an experiment where I ran it on kemerer.arff for 10, 100, 1k and 10k times and recorded both the MMRE and the estimated efforts of projects 13, 14 and 15 (the test 20%), Table 2.

Table 2: Calibrating ngen.

		Estimated (13)	**Estimated (14)**	**Estimated (15)**
	Actual	157	246.9	69.9
ngen	**MMRE**			
10	0.41	55.25	78.88	83.71
100	0.3	102.33	138.76	82.82
1k	0.23	195.33	130.64	77.69
10k	0.189	162.57	81.36	71.3

As expected, the MMRE decreased slower and slower. ngen = 10k took at least half an hour so increasing it wasn't feasible. The estimated efforts for projects 13 and 15 were okay but the estimated effort for project 14 wasn't. To try to fix this, I changed the maximum tree height back to 17 and repeated ngen = 10k. The MMRE was five times better at 0.038 but somehow the estimated costs were worse at 121.86, 97.67 and 83.49 for projects 13, 14 and 15. I changed the maximum tree height, for the last time, back to 6.

I calculated the absolute errors and magnitudes of relative error, Table 3.

Table 3: kemerer.arff results.

ID	13	14	15	
Actual cost	157	246.9	69.9	
Estimated cost	162.57	81.36	71.3	**Mean**
Absolute error	5.57	165.54	1.4	57.5
Magnitude of relative error	0.035	0.67	0.02	0.24

I repeated this for miyazaki94.arff and albrecht.arff, Tables 4 and 5. The actual functions can be seen in the appendix.

Table 4: miyazaki94.arff results.

ID	Q2	R1	R2	S1	T1	
Actual cost	6	5.6	20.1	31.5	50.1	
Estimated cost	15.45	-52.16	17.49	51.23	40.93	**Mean**
Absolute error	9.45	57.76	2.61	19.73	9.17	19.74
Magnitude of relative error	1.58	10.3	0.13	0.63	0.18	2.54

Table 5: albrecht.arff results.

Inquiry,File,…	69,112,…	25,28,…	61,68,…	15,15,…	12,15,…	
Actual cost	61.2	3.6	11.8	0.5	6.1	
Estimated cost	47.52	23.31	26.12	4.03	10.4	**Mean**
Absolute error	13.68	19.71	14.32	3.53	4.3	11.11
Magnitude of relative error	0.22	5.48	1.21	7.06	0.7	2.93

The results were a mixed bag. In kemerer.arff, two of the estimated costs were extremely accurate (within 2-4% of the actual costs) but the third lagged behind. In miyazaki94.arff, the magnitudes of relative error ranged from 0.13 to 10.3 with a mean of 2.54. With the exception of R1, which had a negative estimated cost, I would say the results were okay. In albrecht.arff, bad results were the rule, not the exception.

The question was whether some of the magnitudes of relative error were high because A) genetic programming wasn't an effective way to estimate them or B) it wasn't possible to define one function that worked all of the time. I wanted to explore this further so I tried kemerer.arff again but used projects 4 through 15 to train and 1 through 3 to test. (This is called cross-validation.) The results were worse.

Table 6: More kemerer.arff results.

ID	1	2	3	
Actual cost	287	82.5	69.9	
Estimated cost	56.75	55.98	201.42	Mean
Absolute error	230.25	26.52	131.52	129.43
Magnitude of relative error	0.80	0.32	2.2	1.11

Comparison of results

I used Weka to compare the results with another approach called linear regression. Linear regression tries to find the line of best fit, Figure 2, that models the relationship between a function's input(s) and outputs.

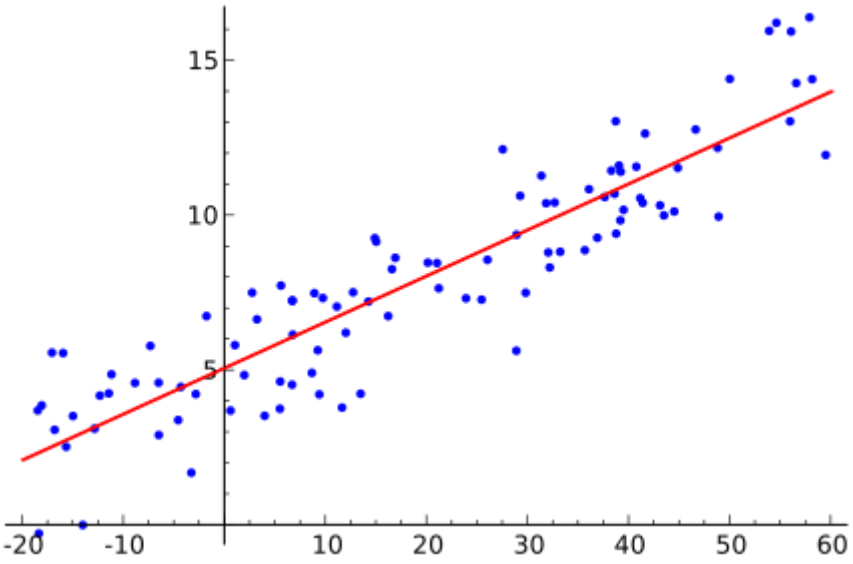

Figure 2: Linear regression.

I configured Weka to run the linear regression algorithm (with default parameters) on each of the three datasets and recorded the results, Table 7.

Table 7: Comparing linear regression and genetic programming.

Dataset	**kemerer.arff**	**miyazaki94.arff**	**albrecht.arff**
Minimum mean absolute error	27.78	17.99	1.27
Max mean absolute error	745.01	450.98	32.49
Mean mean absolute error (LR)	170.77	77.15	9.39
Mean absolute error (GP)	57.5	19.74	11.1

Comparing the mean absolute errors, genetic programming is better than linear regression by about a factor of three in two experiments and about the same in the third. The results aren't enough to say "genetic programming is a better approach to cost estimation than linear regression" but they do suggest it's a feasible approach. Something else that must be taken into consideration is the trade-off between accuracy and speed because the genetic programming approach was a lot slower.

There are lots of ways this work could have been continued. If I had had more time, I would have liked to

- write a better read function that could read all the datasets
- write a cross validation function that could try all the variations
- increase the quality of the code and document it
- experiment with other parameters, not just ngen

Appendix

These are the functions evolved for kemerer…

```
protectedDiv(add(mul(add(add(protectedDiv(KSLOC, ID), sub(ID,
Duration)), mul(Duration, Hardware)), add(protectedDiv(mul(ID, ID),
mul(Language, KSLOC)), sub(protectedDiv(Duration, Language),
Language))), add(add(add(RAWFP, sub(RAWFP, AdjFP)), add(ID, add(ID,
ID))), mul(sub(sub(Hardware, RAWFP), sub(RAWFP, ID)),
protectedDiv(Hardware, mul(Duration, Duration))))), Duration)
```

… miyazaki94…

```
sub(sub(sub(protectedDiv(protectedDiv(sub(EFILE, FILE), sub(KLOC,
SCRN)), ESCRN), neg(KLOC)), protectedDiv(FILE,
add(mul(protectedDiv(SCRN, EFILE), add(FILE, ESCRN)), SCRN))),
protectedDiv(sub(sub(sub(protectedDiv(FORM, EFORM), neg(KLOC)),
protectedDiv(add(ESCRN, FORM), add(SCRN, KLOC))),
protectedDiv(add(add(FILE, KLOC), protectedDiv(EFORM, FORM)),
protectedDiv(sub(ESCRN, KLOC), SCRN))), protectedDiv(add(add(KLOC,
sub(KLOC, SCRN)), protectedDiv(add(FORM, KLOC), sub(SCRN, KLOC))),
sub(protectedDiv(sub(EFORM, FILE), protectedDiv(EFORM, KLOC)), FILE))))
```

and albrecht…

```
mul(mul(File, protectedDiv(mul(sub(sub(AdjFP, FPAdj),
protectedDiv(Input, FPAdj)), FPAdj), add(sub(protectedDiv(AdjFP,
Output), add(Output, Output)), add(protectedDiv(Inquiry, Input),
sub(AdjFP, Output))))), protectedDiv(mul(sub(sub(sub(AdjFP, File),
protectedDiv(AdjFP, Input)), sub(mul(FPAdj, File),
protectedDiv(RawFPcounts, Inquiry))), protectedDiv(mul(sub(RawFPcounts,
File), protectedDiv(AdjFP, RawFPcounts)), add(mul(File, Inquiry),
add(Input, AdjFP)))), add(mul(mul(sub(File, Input), sub(File, Input)),
protectedDiv(neg(Output), sub(RawFPcounts, Input))), add(mul(sub(File,
Input), protectedDiv(AdjFP, RawFPcounts)), add(mul(FPAdj, File),
sub(RawFPcounts, Output))))))
```

And kemerer (2)…

```
add(add(mul(Duration, Hardware), Hardware),
protectedDiv(add(add(sub(protectedDiv(KSLOC, ID), sub(AdjFP, RAWFP)),
sub(add(RAWFP, Duration), sub(AdjFP, RAWFP))), neg(mul(Hardware,
sub(AdjFP, RAWFP)))), sub(protectedDiv(sub(add(Language, Language),
neg(ID)), sub(sub(AdjFP, RAWFP), sub(ID, Hardware))), neg(Duration))))
```