Liam Devlin

Genetic Programming Report

Genetic programming describes the practice of using a genetic algorithm to produce a function / program which has been 'trained' to solve a given problem. The individual takes the form of a tree, which is used as a representation of a function / block of code, which can be compiled and evaluated using error checking metrics.

1. Background of the subject

The task for this assignment was to derive an algorithm that could accurately predict some value in a given data set, using genetic programming. Most GP programs use abstract syntax trees (AST) as the representation of a function during the evolutionary process. ASTs are primarily represented by the terminal set, which is any value or any zero-arity function that produces a value, and the function set, which is the operators and functions available to the GP. Other important characteristics of the tree are the maximum depth of the tree (maximum number of levels down the tree can span) and the parameter set, the data that we pass to the function.

To evaluate the fitness of each individual, a method of compiling the tree in to executable code will be required as to measure the result of the tree. There are many metrics we can use for assessing the fitness of the individual. The primary choice for this program was the Mean Absolute Error or MAE. The mean absolute error is the mean of the differences between the recorded value in a given data set and the predicted value.

Figure 1. Mean Absolute Error formula

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n} |y_j - y_j|$$

The benefits of this technique are that it is relatively fast, easy to understand and generally quite accurate when looking to reduce overall error. It does have some flaws however, if one prediction is wildly different from the recorded value, the MAE might see this large error somewhat mitigated due to the fact that most of the estimates were (relatively) accurate.

To combat this, the Root Mean Square Error (RMSE) should be used. RMSE is similar to the mean absolute error, however the difference between each estimate and answer is squared. This has the effect of steeply lowering the overall fitness of a solution if a large error occurs. All of these differences are then summed and the square root is taken to produce the final value. Using RMSE as a metric will not only reduce the MAE, but should provide a more consistent accuracy with less wild-card errors.

Figure 2. Root Mean Square Error (RMSE) Formula

$$\mathrm{RMSE} = \sqrt{\sum \frac{(y_{pred} - y_{ref})^2}{N}}$$

The final evaluation metric that was considered was the correlation coefficient, which returns a value between -1 and 1, with a negative value representing no correlation and a positive value representing a positive correlation. The closer the value gets to 1, the stronger the correlation between the predictions and the answers. This metric is somewhat useful as it shows that the formula is at least understanding the trend of the data somewhat, however it does not account for the actual values (e.g. [1 ,3, 5] and [10, 30, 50] have a strong correlation but are not similar in terms of their value)

.

Figure 3. Pearson Correlation Coefficient Formula

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

The specific problem we were tasked with was to predict an effort or some equivalent value (effort representing the number of person-hours required for a given project/feature) and compare the GP result with other methods of estimation such as linear regression for multiple data-sets. Each data-set was contained in an .ARFF file, an ASCII document encompassing the data set and an attribute set which defines what each index of each data point represents. Attributes may be either numeric, string, nominal(which behaves similarly to an enumeration) or a date.

2. Implementation

Packages/Libraries used

DEAP - (Distributed Evolutionary Algorithms in Python) is a framework that provides almost all of the functionality needed to set up a GA in python with only the fitness function to be written by the user. It provides an easy-to-use implementation of ASTs which can be evolved as easily as a string or some binary representation.

SciPy & NumPy - Both of these libraries were used when performing more complex numeric operations such as calculating the Pearson Correlation Coefficient.

PyPlot - Used to graph and plot results of each GA run.

Development

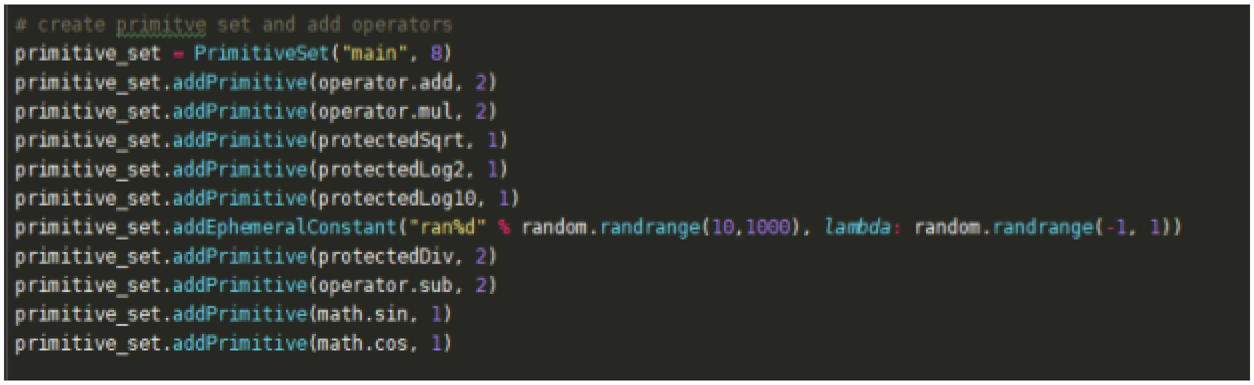
The first step of development was to choose the data sets that would be used for evaluation. The desharnais and CHINA data set were selected and taken from http://tunedit.org/repo/PROMISE/EffortPrediction. It was also important to identify which attributes would be used during the evolutionary process, as including unnecessary information will only serve to clutter the generation and produce worse results than if only the relevant data is supplied.

For the Desharnais data-set, the 'Project', 'YearEnd' and 'PointsNonAdjust attributes were removed. Project and YearEnd were removed as the ID of a project will have no impact on predicting the cost, the same can be said for YearEnd as the year the project ended will also have no impact on the overall cost of a project (disregarding long projects and inflation). PointsNonAdjust was removed as it is simply two other metrics summed together, and is extremely similar to the corrected and more useful PointsAjust attribute.

As the China data-set was slightly larger and had more attributes, slightly more attributes were left out with the hope of significantly increasing the accuracy of the predictions. The decision was made to exclude 'ID', 'Added', 'Changed', 'Deleted' as these are all categorical attributes and again, will have no impact on the overall cost / effort of the project. Furthermore, the "Dev.Type" attribute was removed as it has a value of 0 for every data point, meaning it would not influence the evolutionary process at all. Finally, the N_effort attribute was disregarded as it represents the nominal effort. The decision was made to use the real effort value as the nominal effort is not adjusted for inflation (Investopedia, 2008).

To begin using Genetic Programming with DEAP, we must first define a PrimitiveSet, DEAP's representation of the function set. Figure 4 shows how simple DEAP makes this by allowing each new function to be added with one line of code. Firstly, we define the name of our generated function, in this case "main" as well as passing through the number of arguments for the generated function. This number should be the same as the number of chosen attributes as we will be directly passing the data from each data-point in to the function during evaluation. Each time we add a 'primitive' we tell the PrimitiveSet which function or operator to add, and how many arguments the operation takes. We can also define an Ephermeral constant, defined in the documents as "a no argument function that returns a random value. The value of the constant is constant for a Tree, but may differ from one Tree to another". In this case, we have two possible random selections, a random number between 10 and 1000 or a random number between -1 and 1. This will help in the diversity of random numbers, but is not so large as to begin causing overflow errors.

Figure 4. Function Set definition (PrimitiveSet)

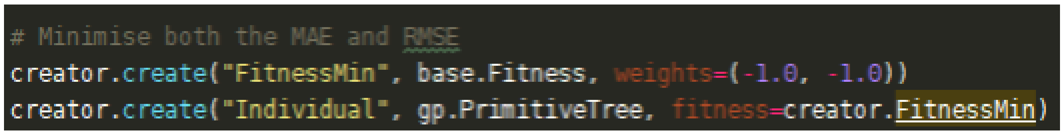


```
# create primitve set and add operators
primitive_set = PrimitiveSet("main", 8)
primitive_set.addPrimitive(operator.add, 2)
primitive_set.addPrimitive(operator.mul, 2)
primitive_set.addPrimitive(protectedSqrt, 1)
primitive_set.addPrimitive(protectedLog2, 1)
primitive_set.addPrimitive(protectedLog10, 1)
primitive_set.addEphemeralConstant("ran%d" % random.randrange(10,1000), lambda: random.randrange(-1, 1))
primitive_set.addPrimitive(protectedDiv, 2)
primitive_set.addPrimitive(operator.sub, 2)
primitive_set.addPrimitive(math.sin, 1)
primitive_set.addPrimitive(math.cos, 1)
```

To study the effectiveness of MAE vs RMSE , the decision was made to perform a single-objective run on each data set, evaluating only the MAE, as well as a multi-objective selection using the MAE and RMSE as equally weighted evaluation metrics. The final individuals' correlation coefficient would also be calculated for each GA run, though retrospectively, the correlation coefficient was not particularly useful to implement.

Firstly, the Multi-objective solution will be demonstrated and is setup identically across both data-sets. Figure 5 shows us defining our creator object, which is used to define an alias of how our GA will work. First we tell DEAP that we will have a fitness function called "FitnessMin", which is derived from the base DEAP fitness class, base.Fitness. The weights represent how important each goal of the GA is to the overall algorithm. As the MAE and RMSE are somewhat similar metrics, it makes sense that we should minimise them both with equal importance. However, if we were to use the correlation coefficient, it might make more sense to use a smaller weighting, thus ensuring that the most appropriate error detection method is being used. We then define an individual, which is of type PrimitiveTree from the GP package of DEAP.

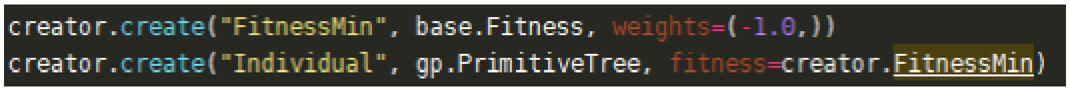
Figure 5. Multi-Objective Creator Object



```
# Minimise both the MAE and RMSE
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)
```

The single objective is very similar, instead only minimising it's first and only objective, the MAE.

Figure 6. Single Objective Creator Object

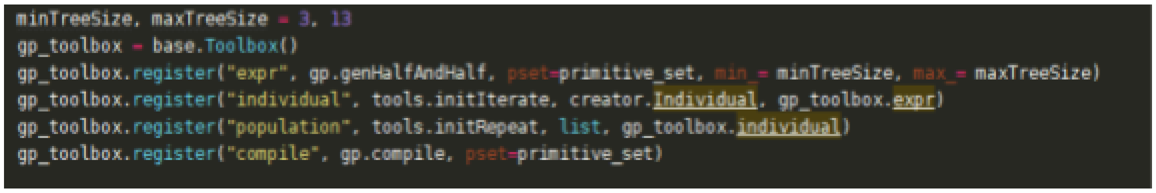


```python
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin)
```

The toolbox is also created in largely the same way across both the multi-objective and single-objective solutions, with the evaluation function being the primary difference. Figure 7 shows the creation of the toolbox, we first must define a limit for the length of each generated tree. This is the limit for each generated member of the population, however it is still possible for a tree to exceed this depth limit through mutation or crossover. This problem is solved by statically limiting the depth of each tree, achieved by attaching a "decorator" to the mutation and crossover stages of evolution, shown in Figure 8. The first step in creating our toolbox is to create a representation of an expression with a given PrimitiveTree, denoted "expr". An expression is created using the primitive set (function set) that was previously defined, setting the minimum and maximum tree size as well as passing through a generation method.

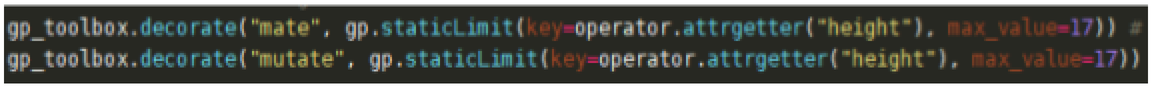
There are three generation methods available to DEAP: Full, Grow and half-and-half; Full will always create a tree where each "leaf" is the same depth, Grow will generate trees where each leaf has a depth somewhere in between the minimum and maximum depth and finally, half-and-half uses a combination of both methods for each leaf. Half and half is utilised as it strikes a good balance between the other two methods, and increases the variation between generated expressions.

Figure 7. Toolbox creation



```python
minTreeSize, maxTreeSize = 3, 13
gp_toolbox = base.Toolbox()
gp_toolbox.register("expr", gp.genHalfAndHalf, pset=primitive_set, min_ = minTreeSize, max_ = maxTreeSize)
gp_toolbox.register("individual", tools.initIterate, creator.Individual, gp_toolbox.expr)
gp_toolbox.register("population", tools.initRepeat, list, gp_toolbox.individual)
gp_toolbox.register("compile", gp.compile, pset=primitive_set)
```

Figure 8. Static Limit of Tree Depth

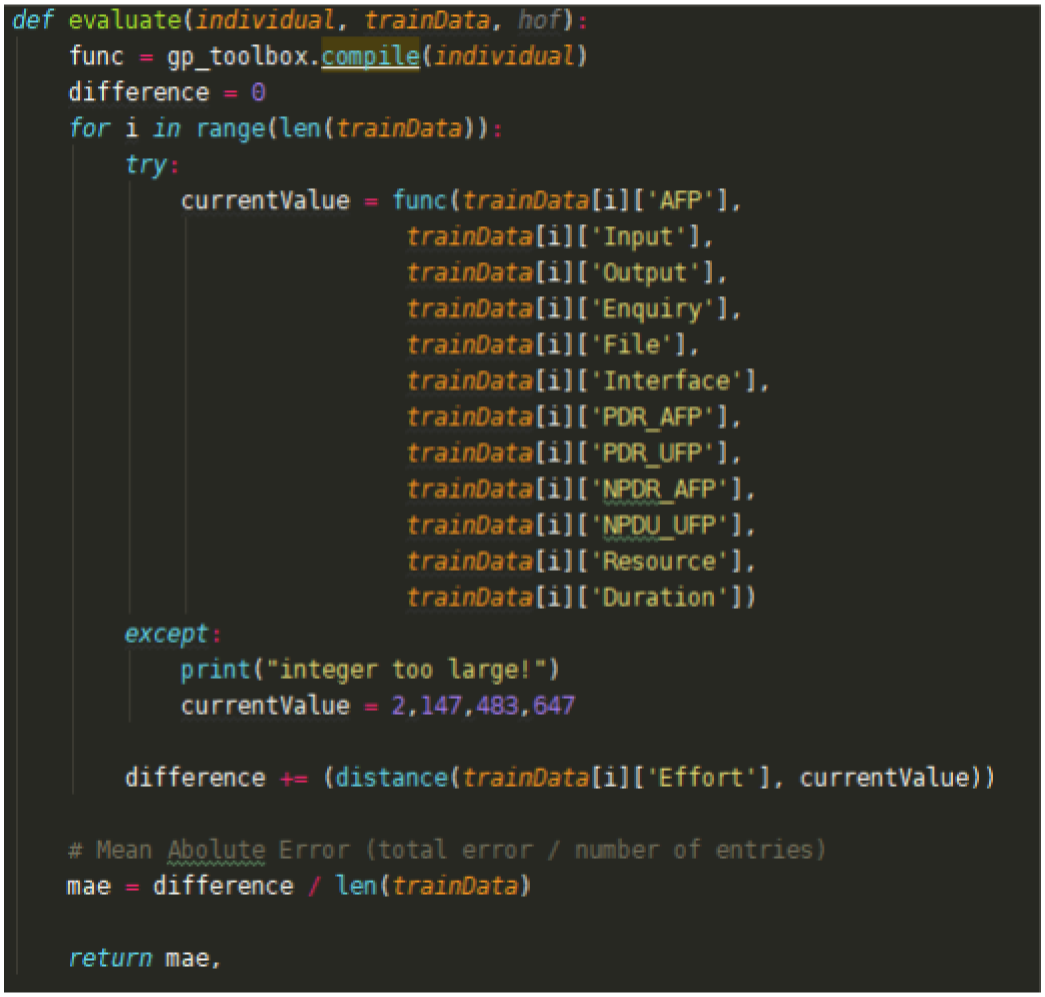


```python
gp_toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height"), max_value=17)) #
gp_toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height"), max_value=17))
```

Next we tell deap that our previously defined alias of an individual (See Figure 5) will be filled out with our newly created "expr" type, the population will therefore become a list of these individuals. The final tool we add to our toolbox is the ability to compile the ASTs we will be generating, this functionality is a built in feature of DEAP and only requires the programmer to pass through the PrimitiveSet that will be used during the evolutionary process.

After preparing the toolbox, the next step was to provide an evaluation function. The single objective evaluation simply evaluates the MAE and returns this as a value to be minimised. As can be seen from Figure 9, we first compile the individual, then iterate through each data point in the training set, evaluating its overall error by adding the distance between the recorded value and the estimation. Once the loop has finished, this difference value is divided by the total number of data points in the set, resulting in the MAE.

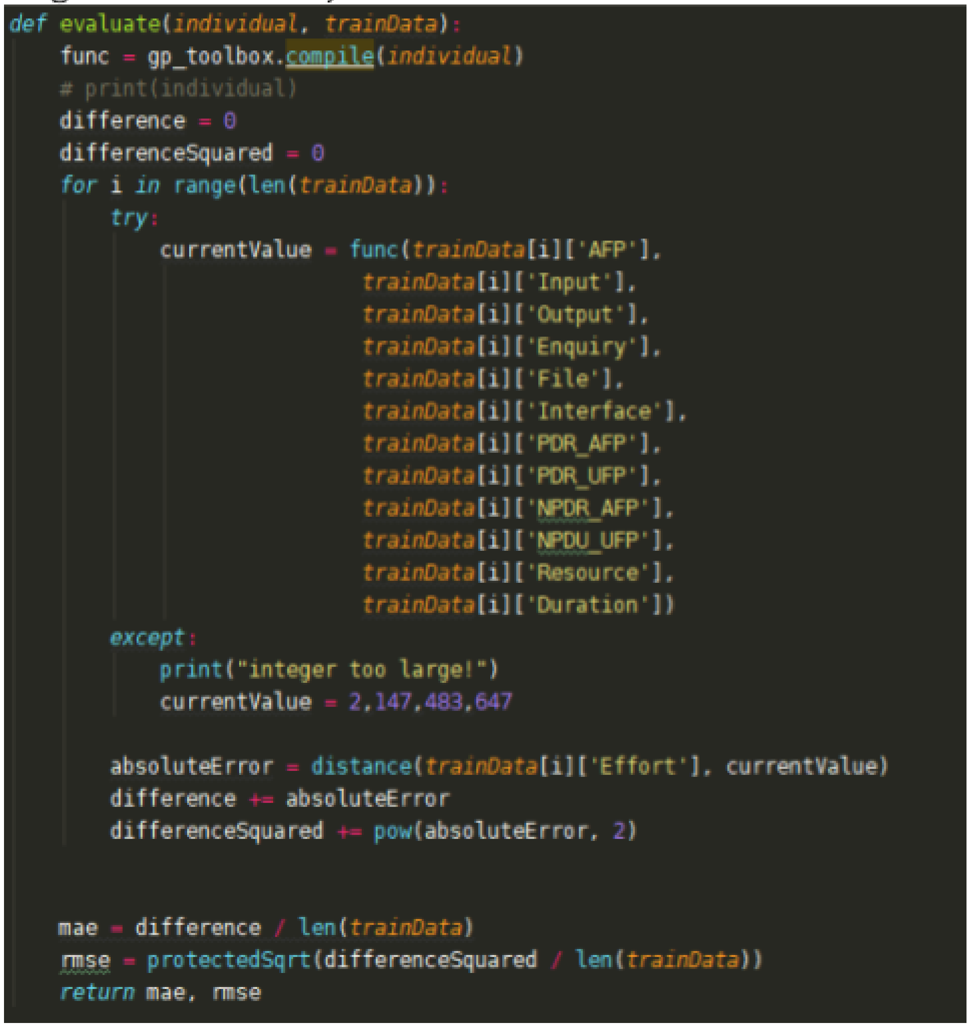
Figure 9. Single Objective Evaluation Method



```
def evaluate(individual, trainData, hof):
    func = gp_toolbox.compile(individual)
    difference = 0
    for i in range(len(trainData)):
        try:
            currentValue = func(trainData[i]['AFP'],
                                trainData[i]['Input'],
                                trainData[i]['Output'],
                                trainData[i]['Enquiry'],
                                trainData[i]['File'],
                                trainData[i]['Interface'],
                                trainData[i]['PDR_AFP'],
                                trainData[i]['PDR_UFP'],
                                trainData[i]['NPDR_AFP'],
                                trainData[i]['NPDU_UFP'],
                                trainData[i]['Resource'],
                                trainData[i]['Duration'])
        except:
            print("integer too large!")
            currentValue = 2,147,483,647

        difference += (distance(trainData[i]['Effort'], currentValue))

    # Mean Abolute Error (total error / number of entries)
    mae = difference / len(trainData)

    return mae,
```

FIgure 10 Shows the multi-objective evaluation function, which shares the same method of calculating the MAE as the single objective solution. The addition of the RMSE calculation is relatively straight-forward as it is similar to the MAE. To calculate the RMSE, we again iterate through every point in the training set and find the distance between the estimate and the recorded value. This value is squared and added to a counter variable. Once the loop has completed we take the square root of this value divided by the total number of data points.
As can be seen from both Figure 9 & 10, we pass through each selected attribute as a parameter for the newly compiled function.

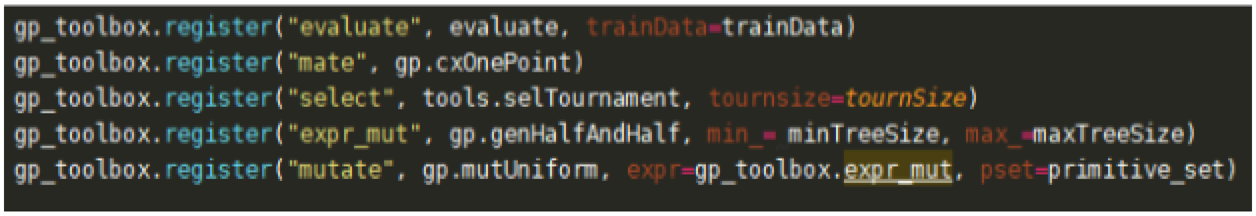
Figure 10. Multi Objective Evaluation Method



```
def evaluate(individual, trainData):
    func = gp_toolbox.compile(individual)
    # print(individual)
    difference = 0
    differenceSquared = 0
    for i in range(len(trainData)):
        try:
            currentValue = func(trainData[i]['AFP'],
                                trainData[i]['Input'],
                                trainData[i]['Output'],
                                trainData[i]['Enquiry'],
                                trainData[i]['File'],
                                trainData[i]['Interface'],
                                trainData[i]['PDR_AFP'],
                                trainData[i]['PDR_UFP'],
                                trainData[i]['NPDR_AFP'],
                                trainData[i]['NPDU_UFP'],
                                trainData[i]['Resource'],
                                trainData[i]['Duration'])
        except:
            print("integer too large!")
            currentValue = 2,147,483,647

        absoluteError = distance(trainData[i]['Effort'], currentValue)
        difference += absoluteError
        differenceSquared += pow(absoluteError, 2)


    mae = difference / len(trainData)
    rmse = protectedSqrt(differenceSquared / len(trainData))
    return mae, rmse
```

After defining our evaluation method, the final step is to finish our toolbox and run the GA. As can be seen from Figure 11, the evaluation function is registered to the toolbox, and the programmer selects what methods will be used for crossover, mutation and selection. Crossover is a standard one-point crossover, the selection method used is tournament selection and was chosen because of its ability to work well in both single and multi-objective contexts. Mutation is slightly different from other representations as we need to explicitly tell DEAP how to mutate an AST. To do this a helper function is created named "expr_mutation" which will be used by the mutation function proper. This simply passes through the half-and-half generation method with the minimum and maximum tree-size. Finally, we define our mutation function using uniform mutation, and pass through our previously defined expr_mutation, which tells the uniform mutation method how to mutate a given tree.

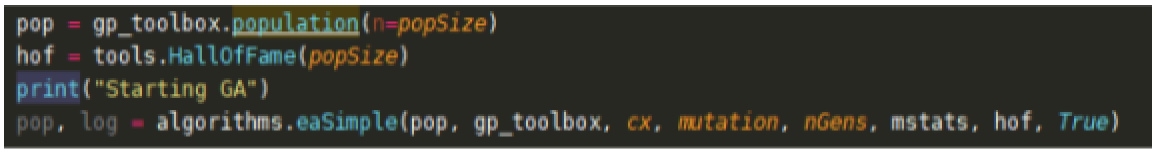
Figure 11. Toolbox Creation Contuned



```
gp_toolbox.register("evaluate", evaluate, trainData=trainData)
gp_toolbox.register("mate", gp.cxOnePoint)
gp_toolbox.register("select", tools.selTournament, tournsize=tournSize)
gp_toolbox.register("expr_mut", gp.genHalfAndHalf, min_= minTreeSize, max_=maxTreeSize)
gp_toolbox.register("mutate", gp.mutUniform, expr=gp_toolbox.expr_mut, pset=primitive_set)
```

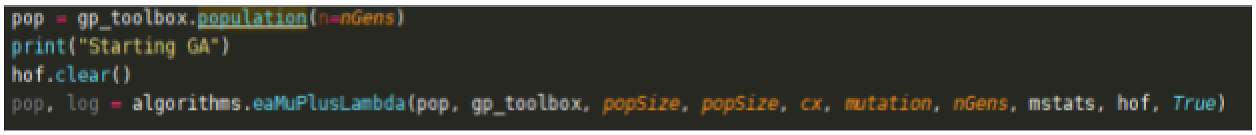
After the toolbox has been setup we can begin our GA. For single-objective runs the eaSimple method was used as it is fast and most appropriate for GAs with one objective. For the multi-objective runs eaMuPlusLambda was used, as this allows us to consider a small number of invalid candidates, increasing overall variety of the population, and minimising selection bias. The return of each GA method is an optimised population and a statistical logbook containing information about each generation of that run.

Figure 12. Single Objective Evolutionary Method



```
pop = gp_toolbox.population(n=popSize)
hof = tools.HallOfFame(popSize)
print("Starting GA")
pop, log = algorithms.eaSimple(pop, gp_toolbox, cx, mutation, nGens, mstats, hof, True)
```

Figure 13. Multi Objective Evolutionary Method

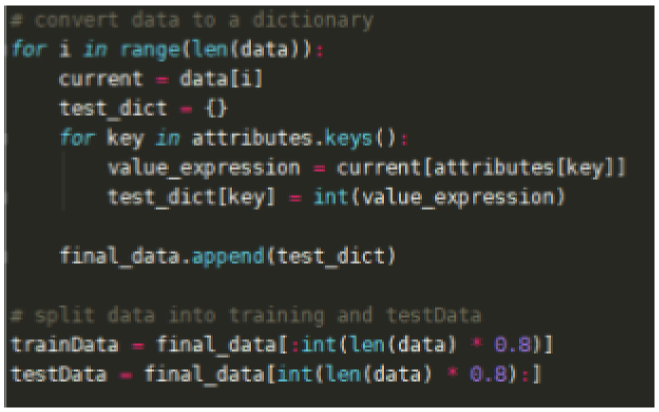


```
pop = gp_toolbox.population(n=nGens)
print("Starting GA")
hof.clear()
pop, log = algorithms.eaMuPlusLambda(pop, gp_toolbox, popSize, popSize, cx, mutation, nGens, mstats, hof, True)
```

Test / Training Data Split

Both data-sets were evaluated using an 80/20 split of the data, with 80% for training and a remaining 20% for testing. This was achieved by collecting all the information into a dictionary and using python magic to easily split up the data. As can be seen from Figure 14, we collect all of the identified appropriate attributes from the final data set, using a custom dictionary containing only the names and index of the identified useful attributes. Once the full data set has been organised, it is split using the python [:] operator which return all elements before or after a provided index.

Figure 14. Train / Test data split



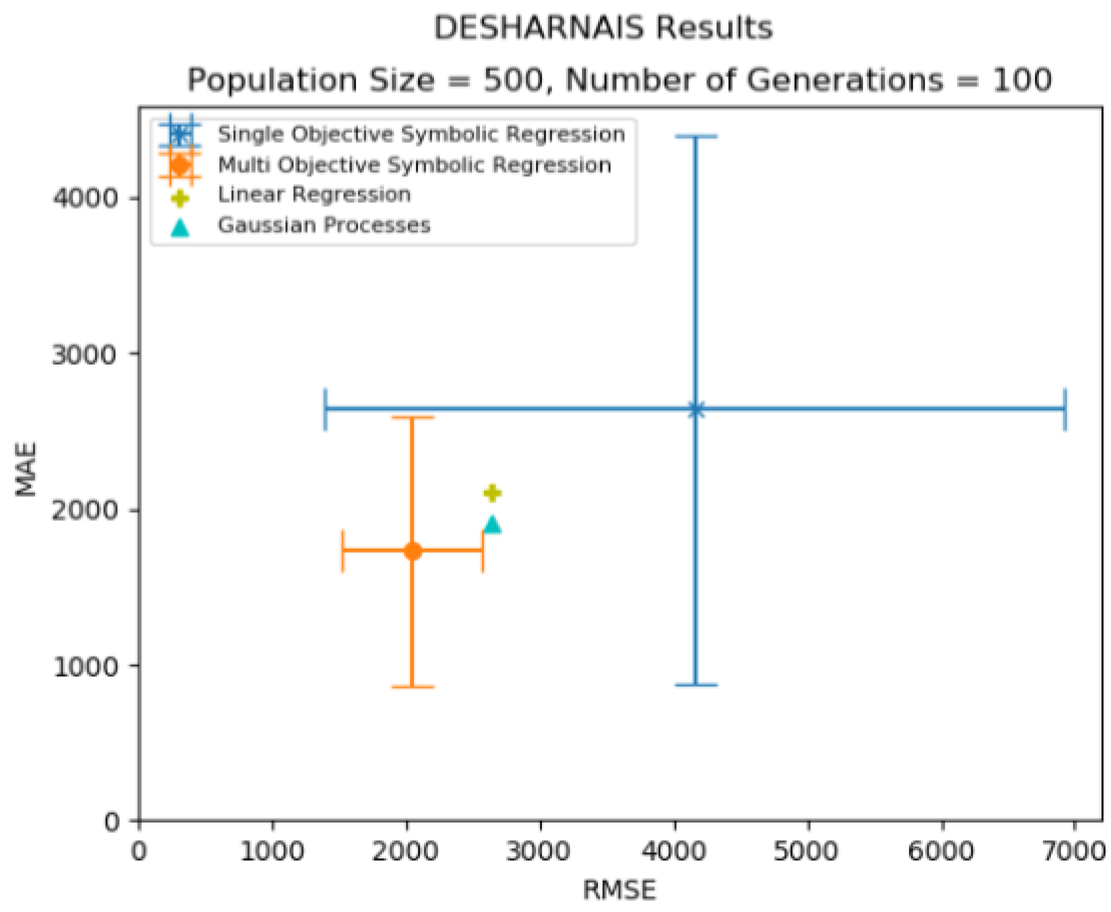
```
# convert data to a dictionary
for i in range(len(data)):
    current = data[i]
    test_dict = {}
    for key in attributes.keys():
        value_expression = current[attributes[key]]
        test_dict[key] = int(value_expression)

    final_data.append(test_dict)

# split data into training and testData
trainData = final_data[:int(len(data) * 0.8)]
testData = final_data[int(len(data) * 0.8):]
```

3. Results

Across all runs of the GA, the mutation rate, crossover rate, population size, number of generations and tournament size were kept identical.

Desharnais results

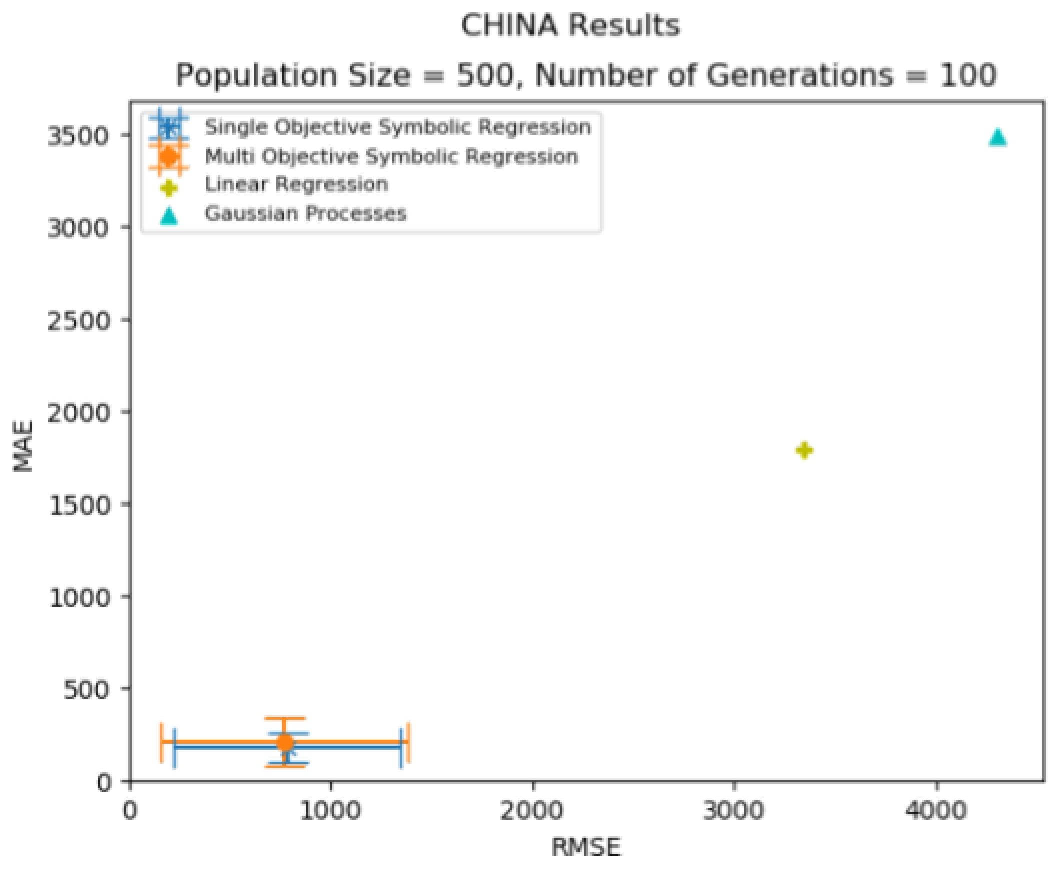
Figure 15. Desharnais Final Results



Overall the desharnais results are somewhat surprising, with the single objective solution scoring notably worse than linear regression and the gaussian process. The multi-objective however scores better than both linear and gaussian. This highlights the previously mentioned limitation of the MAE, as it is not prone to picking up large errors like RMSE. This is demonstrated by the massive spread of the single-objective solution, compared to the relatively low spread of the multi-objective solution. To combat this problem in the future, the RMSE may be used as the main method of evaluation instead of MAE.

China results

Figure 16. China Final Results



The china results were much more in-line with what was expected, with both the single and multi objective solutions being far superior to both linear regression and the gaussian process. It is surprising to see that the multi-objective and single-objective solutions are so close to one another, considering the results of the desharnais GP. This may in fact highlight a potential issues with the way data-splitting is handled. As this system used a basic 80/20 split instead of cross-validation, there is a heavy bias towards the training set, the implementation of a cross-validation method may see an increase in the consistency between different data-sets.

Overall conclusions

Overall Multi-Objective GP was the clear winner, with single-objective GP proving to be slightly more inconsistent, at least with a traditional 80/20 data split procedure. It is easy to see why Genetic Programming is becoming more widely-used due to the potential for vastly increased accuracy compared to traditional methods of cost-estimation.

References

, I. (2008, April 18). Nominal Vs. Real GDP, And The GDP Deflator. Retrieved November 26, 2018, from https://www.investopedia.com/exam-guide/cfa-level-1/macroeconomics/nominal-real-gdp-deflator.asp