caRtesian – Getting Started

Author: Ryan Porteous
rp10@hw.ac.uk

The software package caRtesian provides an implementation of Cartesian Genetic Programming directly in R. The package is to be used in a data mining environment for symbolic regression problems. This guide aims to introduce the package and how to get started using it.

Introduction to Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a variety of Genetic Programming which uses graphs to represent programs instead of the traditionally used trees. It creates a population of solutions to a given problem and progressively improves these solutions through an evolutionary process.

The caRtesian R Package

The package is used to solve symbolic regression problems where a dataset is provided containing rows of input values and the desired output from these values, but the function providing the output value is unknown. The package provides generates candidate solutions for this function and assesses each of them with the desired output values in the dataset. The solutions progressively improve through a process called evolution as the program runs.

Installation

The package is available on GitHub and can be installed by using the devtools package and running the following command in an R session:

```
> devtools::install_github(“porteous54/caRtesian”)
```

As the package is installed, the dependencies are also installed so installation can take longer than expected if the dependencies are not already installed.

Getting Started

The package must be loaded into the running R session can be done through the command:

```
> Library("caRtesian")
```

Loading Variables

The next step is to load the required variables into the Global Environment. The variables that are needed, are a `dataset` containing the input and output values, a `model` specifying the input and output fields of the dataset, and the set of functions that the program can sample from to build a solution called a `functionSet`.

For this example, the `dataset` will be `x_squared_minus_y` which contains two input values `x` and `y`, an output value: `output`. The output value is actually the result of the function $x^2 - y$ but the program does not know this and will try its best to find this function. The dataset can be loaded using the command:

```
> dataset <- x_squared_minus_y
```

The dataset structure and example values can be viewed using the command:

```
> head(dataset)
```

This shows the name of each column and shows the values contained in it. The next step is to define a model which describes this dataset. Since this dataset was created for the purpose of this guide, the `output` depends on both the `x` and `y` variables, but this is not always the case in a symbolic regression problem. The model can be defined through the command:

```
> model <- output ~ x + y
```

Finally, the set of functions must be loaded and in this case a set containing simple arithmetic operations will be used. This can be done through the command:

```
> functionSet <- arithmeticSet()
```

The functions contained in this set can be viewed by writing the name of the variable `functionSet` into the console:

```
> functionSet
  funcName arity
1        +     2
2        -     2
3        *     2
```

Run the Program

To begin running the program and start searching for solutions we use the `cgp` function and give the variables we have defined already as arguments to it as well as adding some others:

```
> set.seed(10)
> result <- cgp(dataset = dataset,
+               model = model,
+               functionSet = functionSet,
+               maxGenerations = 15,
+               rowsFuncNodes = 6,
+               colsFuncNodes = 2,
+               levelsBack = 1,
+               updateFreq = 3)
```

The `set.seed` function is used so that the results will be reproducible. Then the program is run which runs for a maximum of 15 generations but will finish before that if it finds a solution. The rest of the parameters are to define the shape of the graph used inside the program to represent the solution.

As the program runs, details of its progress are displayed in the console with the best solution found printed as a mathematical equation at the end of execution. An example of this can be seen here:

```
Generation: 15 / 15
Fitness of best solution so far: 52.159
Average fitness of population: 704.783

Best solution found as text:
(a * (c + a))
```

View the Results

Once the program is finished, the results are stored in the variable we defined called `result` which contains various information so that the solution found can be used now with future data. The progress information that is printed as the program runs is also stored in the result so that it can be viewed as a plot. The plot can be loaded through the command:

```
> plotGraph(result$plotData)
```

The graph shows the how the solution evolved over time and hovering over each of the points provides the exact value of the fitness and generation. There are tools attached such as zooming and being able to select a subset of points for a closer display of them.

In this case the program was unable to find the exact function that the dataset was formed but there are actually multiple parameters that can be tuned to suit the needs of the user. These parameters are discussed in the next section.

Changing the Parameters

The parameters used in the previous section produced a solution which was not correct but was the best found. It is possible that by tuning these parameters a perfect solution will be able to be found. The parameters set previously: `rowsFuncNodes`, `colsFuncNodes` and `levelsBack` affect the structure that it is possible for the solution to take so it may be the case for a certain dataset, these parameters simply cannot allow the solution to be represented fully.

There are also hidden parameters that have not been seen yet which are `fitnessFunction` and `selectionMethod`. These parameters have default values so that users less experienced with Genetic Programming can still get started in using the package.

Changing the Fitness Function

The `fitnessFunction` parameter specifies the function used to assess a solution against the desired output defined in the dataset. There are two fitness functions distributed with the package which are `mae` (Mean Absolute Error) and `rmse` (Root Mean Squared Error). The default fitness function is `mae`, but this can be changed to `rmse` through changing the `fitnessFunction` parameter to `rmse`. Parameterising this also allows any user of the package to define their own fitness function and also pass it into the program, as long as it accepts the same arguments as `mae` or `rmse` and returns a numeric value.

Changing the Selection Method

The `selectionMethod` parameter expects a list which provides the function to use for selection, and its associated arguments. The program by default uses the `muLambdaStrategy` which accepts three parameters so these must be defined in the list. An example of how `muLambdaStrategy` is used is shown below.

```
selectionMethod = list(func = muLambdaStrategy,
                       args = c(population = NA, 4, NA))
```

The vector `args` follows the same order that the `muLambdaStrategy` function accepts and dummy values are provided here for the `population` and the last parameters, `functionNodeStructure` since these change during execution of the program.

Another selection method called `tournamentSelectionStrategy` is also distributed with the package. To use this selection method instead, the `selectionMethod` must be changed but it still follows the same structure since `tournamentSelectionStrategy` accepts the same parameters.

```
selectionMethod = list(func = tournamentSelectionStrategy,
                       args = c(population = NA, 4, NA))
```

To help with structuring this parameter, there is a function `validSelectionInput` which checks the structure matches what the program expects.

Using the Solution with New Data

The solution is stored in the result of the program, along with the set of functions the program used to sample from when creating the solution. This is so that new data can be entered into the solution and a value can still be given. Data can be loaded into the new solution with the command:

```
> result$bestSolution$inputNodes$value[x] <- value
```

In the above command `value` is the value to be loaded and `x` is the row to load it into. Be careful not to overwrite the last row as this stores a random constant value used by the program.

An example which can be used with the result generated in the Getting Started section is:

```
> result$bestSolution$inputNodes$value[1] <- 5
> result$bestSolution$inputNodes$value[2] <- 10
```

To then get a value using the new loaded data use the command:

```
> decode2(result$bestSolution, result$functionSet)
```