# Predicting forest cover types from cartographic variables via a genetic programming approach

Wim Looman
Electrical and Computer Engineering
University of Canterbury
Christchurch, New Zealand
wgl18@uclive.ac.nz

Steve Webb
Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
steve.webb@gmail.com

***Abstract*—Blah blah blah.**

## I. Introduction

This report details the implementation and testing of a genetic programming algorithm in Ruby with the goal of learning to predict forest cover types from a set of artographic variables.

### A. Genetic Programming [1]

Genetic programming (GP) is a form of evolutionary computation (EC), one of the major subsets of machine learning. One of the major advantages of genetic programming over other methods is that the user does not need to specify any sort of form or algorithm for the computer to use as a basis to solve the given problem. Instead the user simply has to supply a *fitness function* that can be used to rank solutions, the algorithm then naturally evolves as part of the learning process.

*1) The Basics:* The basic outline of GP is similar to other forms of EC, you start with an initial randomised population, perform some method to select a good subset of the population, then use this subset to generate a new generation of the population. In GP the population is made up of a collection of programs/algorithms, these are generally tested against a given fitness function and then a stochastic selection process finds the best and does the transformation to the next generation.

Looking at this description the first decision that needs to be made is how to represent the program. There are generally two solutions to this, based off how the simplest of current programming languages work, either a linear stack based approach or a tree based approach.

Out of the two approach the linear stack based approach is definitely easier to implement, especially when creating the algorithm for transforming one generation to the next. Unfortunately it is a very rare approach in modern programming languages, likely because it is a lot harder to write easy to understand, decomposable code with. Other than esoteric languages such as Befunge or Golfscript the most widely used form of this would have to be the many different forms of assembly code. It is also almost completely absent in literature of the subject, this is disappointing as I believe that it could be a very effective avenue.

The tree based approach is more similar to current languages, it easily allows representation of code such as `[IF (a == 0) THEN (2) ELSE (4) END]` as a simple six node tree, see Figure 1. This provides a few problems when attempting to construct a new generation, but nothing that can't be overcome.

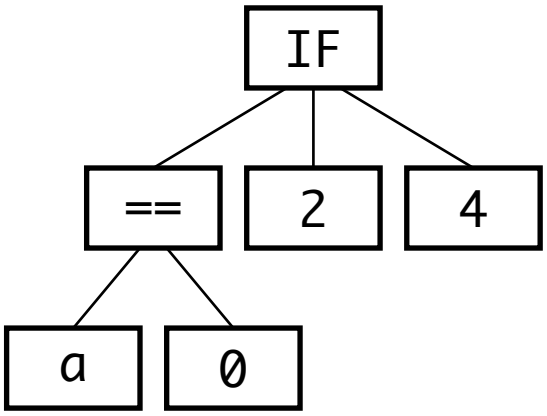


Figure 1. A simple tree representing `[IF (a == 0) THEN (2) ELSE (4) END]`.

The decision of what possible nodes to have in the tree is very important, generally there are three types of nodes; function nodes, variable nodes and constant nodes. Variable and constant nodes are only leaf nodes while function nodes are all the non-leaf nodes. The purpose of these nodes is completely defined in their descriptions, variable nodes contain the input variables of the algorithm, constant nodes contain constants for comparison against the variables and function nodes contain functions such as `IF` and `==` as seen above.

The selection process in all EC methods is very similar, individuals are selected by a probabilistic

function based on their fitness. The most common method employed in GP is *tournament selection*, this is where a subset of the population is randomly selected and sorted in order of fitness. The top `n` individuals are then chosen as required by the crossover or mutation operators.

The last part of the process is how to perform crossover and mutation, this is where GP differs the most from other evolutionary algorithms. The most common form of crossover used in GP is *subtree crossover*. Two parents are selected then a random node in each is chosen, the first parent is then copied and the chosen node is replaced with the chosen node from the second parent.

For mutation the most commonly used operation is *subtree mutation*. This is when a random point in the individual is selected then replaced with a new randomly generated subtree. To try and keep a semi-consistent size the generated subtree is kept to approximately the same height as the removed subtree.

### B. Dataset

The dataset used came from the US Forest Service inventory information and information derived from standard digital spatial data processed in a geographic information system as gathered by Jock A. Blackard and Denis J. Dean [2].

The classification variable is the forest cover type, this is one of seven mutually exclusive classes. The cover type maps were created by the US Forest Service and derived from large-scale aerial photography.

The rest of the variables were derived from digital spatial data obtained from the US Geological Survey and the US Forest Service. The variables used were:

1. Elevation (m)
2. Aspect (azimuth from true north)
3. Slope (°)
4. Horizontal distance to nearest surface water feature (m)
5. Vertical distance to nearest surface water feature (m)
6. Horizontal distance to nearest roadway (m)
7. A relative measure of incident sunlight at 09:00 h on the summer solstice (index)
8. A relative measure of incident sunlight at noon on the summer solstice (index)
9. A relative measure of incident sunlight at 15:00 h on the summer solstice (index)
10. Horizontal distance to nearest historic wildfire ignition point (m)
11. Wilderness area designation (four binary values, one for each wilderness area)
12. Soil type designation (40 binary values, one for each soil type).

All numeric values in the dataset were mapped into the range (-1,1), this makes ensuring that any constants generated are in the right range a lot easier.

The dataset was then split into three mutually exclusive and distinct sets, a training set, validation set and test set. The full data set contained 581 012 observations, the training set was composed of 1620 randomly selected observations from each cover type (11 340 observations total), the validation set was composed of 540 randomly selected observations from each cover type (3780 observations total) and the test set was the rest of the observations (565 892).

These sets were chosen to match the original study by Blackard and Dean, this way the results of this study could easily be compared to that study.

## II. Implementation

### A. Plan of Action

The plan was to create a simple general GP system in Ruby, then attempt to use the system to solve the chosen dataset. By building a general system the testing of the system will be greatly simplified, a nice easy to learn function such as `x^2 + y^2` can be used to verify the system. Using Ruby provides a nice easy interface to define the available functions, constants and variables via meta-programing.

The first step in the program is the definition of the functions, constants and variables. Instead of hard-coding it was decided to create a simple domain specific language (DSL) to allow them to be programmaticly defined at startup. This will then be saved into an environment variable that will be added to the required classes via meta-programing.

The individual algorithms will be stored as a tree, with each node containing a representation of the basic code for the node. To run the algorithm the tree can be flattened by with each node's textual representation having the children nodes substituted in recursively till just a single string representing the function is returned. This string can then simply be eval-ed by Ruby and the output checked against the fitness function.

The population can be stored as a simple array of the individuals contained, Ruby array's come with a `sample` method that will easily allow a random subset of the population to be gathered for use in a tournament.

Algorithms will also have support for the basic operations such as *mutation* and *crossover* added. Another operation yet to be described that will be supported is *simplification*, this will take an existing algorithm and simplify any constant parts of the algorithm. The random mutation and crossover regularly creates bits of code like `if (false) then (1 - (4 * 8)) else (4 + (9 - ((12 - 6) / 6))) end`, this can very easily be simplified down to just `12`.

### B. Solution

The DSL was created in a `Builder` class. The constants and variables are simply defined inside a YAML dictionary, variables just a mapping from variable name to variable type, e.g. `elevation: Number` for the *elevation* variable which is a *Number* type, constants are a mapping from the constant type to a piece of Ruby code to generate a new random constant, e.g. `Boolean: (rand > 0.5)` for generating a *Boolean* constant using the inbuilt Ruby `rand` function.

Functions are a little more complex, a simple syntax was created that could be parsed using just regular expressions and the required functions could be easily created from. A simple example of this syntax is the `PLUS` function with the definition:

```
PLUS: Number, Number -> Number
 => ({0} + {1})
```

This says that `PLUS` is a function that takes as input two `Number` s and gives as a result a `Number`, the Ruby code for the function is then given with `{n}` used to embed the nth argument.

A more complex example is the definition of the `IF` function:

```
IF: Boolean, <X>, <X> -> <X>
 => (if {0} then {1} else {2} end)
```

The main difference with this is the use of generics, any type that matches the regex `<.+?>` is counted as a generic type. During generation of the allowed functions these are replaced with each of the constant types defined to produce a set of functions. So this `IF` function takes in a `Boolean` argument, then 2 arguments of the same type and returns a result with the same type as the last two arguments. Again the Ruby code to define this functions is given last.

As well as the definitions the `Builder` is responsible for gathering other configuration parameters such as; population size, crossover/mutation/simplification/reproduction rates and max initial algorithm height.

Once the environment has been set-up using the `Builder` the initial population has to be produced. This is achieved using a technique called *ramped half-and-half* [3]. This is a technique where algorithms are generated using half *grow* and half *full* with the maximum algorithm height being slowly increased as the population is created. This is in order to create an initial population with a variety of algorithm sizes and shapes.

Next the population has to be evolved. This is done in the `Population` class, `Array#sample` is used to pick out a tournament of individuals to compare. These are then sorted by their scores and the best one or two as required are picked off for the mutation and crossover operations. This is repeated until the new population is at the required size and this is then returned.

The actual scoring is done inside the fitness function defined by the user. This calls the algorithm with the required variables and compares the result to what is expected. Because of using a series of examples in the fitness function this was set-up as a lower is better scoring system. Each example that was incorrectly classified was worth one point, each example that was correctly classified zero and the average height of the algorithm's tree was worth 0.1 per layer.

### C. Implementation Issues

The biggest issue with this implementation was speed, Ruby is definitely not the fastest of languages. Initially I believed the ease of implementation and understandability of the code would out-weigh the speed disadvantage, but after seeing how slow it actually is I now think a better method would be to build a hybrid C/Ruby system.

The other major issue didn't arise until I attempted to classify the actual dataset. When learning the test set of just `x^2 + y^2` the best solution would generally be found within 3-7 generations, around 1-3 minutes of computation. After verifying that it was running with the actual test set I spawned four separate populations and left them running for a few hours. Checking on them later I was surprised to see them sitting at approximately 60GiB total memory usage. I'm still not sure whether this was a problem with my code, or if I had just hit one of the known memory leaks in Ruby. One issue with using a new language with a problematic garbage collector.

### D. Solutions

The solution to both the above issues was the same: distributed computing. There is a really nice background job queueing library called Resque[1]. By using this with a Redis[2] database I could easily distribute the computation around a few hundred computers on the university network.

This also solves the second problem via Resque's architecture; each worker uses a parent-child pair. The parent simply loads the environment and waits for a job, once a job arrives the parent forks a new child that does the processing, when the child finishes it exits, freeing up the memory used during the run.

The slowest part of the system was calculating scores for each individual algorithm; so the simplest way to distribute the system was to save each population into a Redis list, have the workers pull individuals from this list, compute their scores and push them into a new list. Once the old list was empty one worker pulls the entire new list in, creates the new generation, and pushes the new generation back into the old list.

[1] https://github.com/defunkt/resque

[2] http://redis.io

Because these were University computers I couldn't simply leave them all logged in so the system had to be resilient to workers suddenly disappearing when others used the computers. Each individual wasn't very important overall so if we lost a few when a worker dissapeared this shouldn't affect the system. Therefore we could simply have each worker only pulling one individual out of the database at a time, then put it back in once scored before pulling the next out. If we lose any then the population size will be built back up when the next generation is created.

The biggest problem would be if a worker crashed when creating the new generation, at this point the worker has to have the entire population pulled out to compare them all. There were two parts to the reliability here; first, the workers that could create the new generation were limited to only those on the main computer science server; second, as each individual is pulled out of the database a copy is pushed into a backup list.

Keeping the workers on the main server would probably be enough by itself. This is the same server that the Redis database is running on and Redis is a purely memory-based database, so if the server were to crash everything would be lost anyway. The backup is more if the worker hits some strange error and gets stuck. If so the backup can be manually copied across into the old population list and the job restarted.

The transformation of the system from a single-threaded, single-process system into a distributed system was rushed and turned out a big hack, but it worked. At its peak I was using approximately 440 workers distributed over around 130 computers.

## III. Results

## IV. Conclusion

### A. What I Learned

### B. Future Work

## References


[1] R. Poli, W. B. Langdon, and N. F. McPhee, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008, (With contributions by J. R. Koza). [Online]. Available: http://www.gp-field-guide.org.uk

[2] J. A. Blackard and D. J. Dean, "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables," *Computers and Electronics in Agriculture*, vol. 24, no. 3, pp. 131 – 151, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168169999000460

[3] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.