Journal of Statistical Software
July 2016, Volume 71, Issue 1. doi: 10.18637/jss.v071.i01

gramEvol: Grammatical Evolution in R

Farzad Noorian
The University of Sydney

Anthony M. de Silva
The University of Sydney

Philip H. W. Leong
The University of Sydney

Abstract

We describe an R package which implements grammatical evolution (GE) for automatic program generation. By performing an unconstrained optimization over a population of R expressions generated via a user-defined grammar, programs which achieve a desired goal can be discovered. The package facilitates the coding and execution of GE programs, and supports parallel execution. In addition, three applications of GE in statistics and machine learning, including hyper-parameter optimization, classification and feature generation are studied.

Keywords: evolutionary algorithms, optimization, grammatical evolution, R.

1. Introduction

Grammatical evolution (GE, O'Neill and Ryan 2001) generates complete programs, optimized towards performing a certain task by combining context-free grammars (CFG, Knuth 1964) and genetic algorithms (GA, Holland 1992). Specifically, syntactically correct programs are generated from a user-defined grammar, using a binary string to choose grammatical production rules. Through a formulation involving a user-defined cost function, the fitness score of programs for solving a problem can be evaluated, and an optimization process is used to search through a subspace for the best program. This optimization's objective function, i.e., mapping a binary string to a program and subsequently to a numeric score, is often non-smooth and non-convex, precluding gradient-based optimization algorithms and favoring evolutionary optimization techniques such as GA.

GE is an alternative to genetic programming (GP, Koza 1992) for generating programs via evolution. While GP directly operates on the actual program's tree structure, GE applies evolutionary operators on binary strings which are subsequently converted to the final program. GP normally requires a custom search strategy to generate correct programs, whereas GE can utilize an unconstrained evolutionary search, relying on the mapping to generate correct programs.

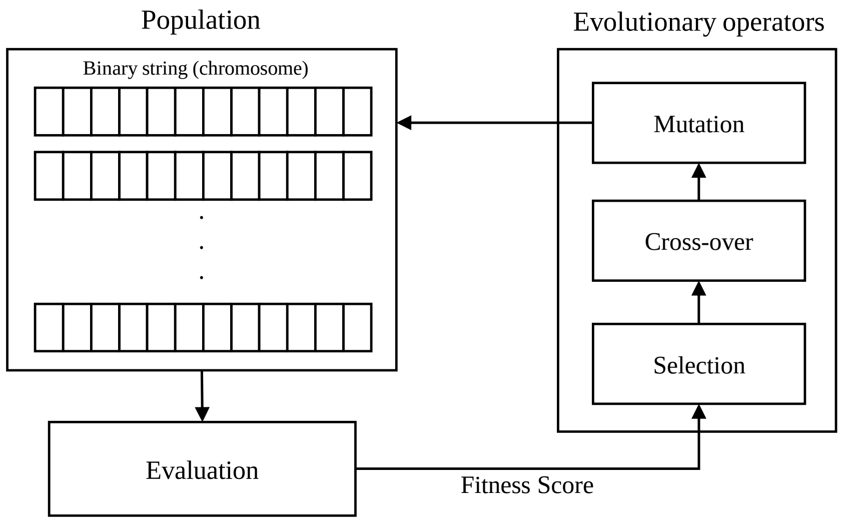

Figure 1: The evolutionary process and associated operators.

GE allows quick and easy integration of domain specific knowledge into the optimization problem through a customizable grammar. It has been successfully applied to many research areas in science and engineering, including computational finance, smart grid forecasting, music, and robotic control. A survey by McKay, Hoai, Whigham, Shan, and O'Neill (2010) discusses the range of GE research and applications.

In this paper, we present the R package **gramEvol** (Noorian and de Silva 2016), which facilitates the construction and execution of programs in R (R Core Team 2016) using GE. The rest of the paper is structured as follows. In Section 2, evolutionary algorithms with emphasis on GE are briefly studied. Section 3 introduces the package and describes its functions. Finally, in Section 4, three example problems are analyzed and solved using GE. It is shown how a grammar can simplify model selection, hyper-parameter optimization, classification, and feature generation.

2. Background

2.1. Canonical genetic algorithms

Canonical GA (Holland 1992) is an optimization algorithm which operates on a *population* of *chromosomes*, performing evolutionary operations including selection, crossover, and mutation as illustrated in Figure 1. Inspired by biological evolution, GA has been successfully used in applications with complex fitness landscapes and multiple local optimas (Mitchell 1996).

In canonical GA, a chromosome is represented by a binary string. Normally, modern GA implementations do not directly operate on binary values. Instead, bits are grouped into n-bit values creating a *codon*, each of which is used as a parameter in the optimization problem. If the problem is made of multiple building blocks, codons related to each block are grouped together as a *gene* (Figure 2). This arrangement is a logical presentation of data and does

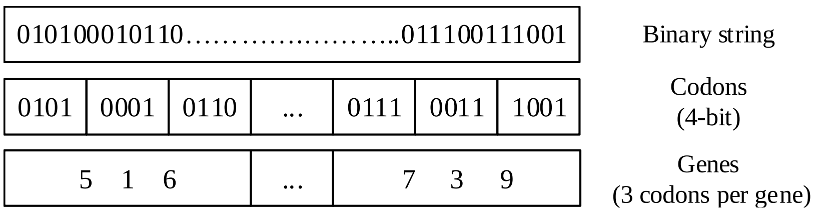

Figure 2: Chromosome representations in GA.

not affect the low level representation of the chromosome.

The initial population is created from randomly generated chromosomes, each representing a solution to the formalized problem. The chromosomes are then evaluated based on a given *cost function* ϕ. The objective is to minimize the cost function, or maximize the *fitness* of the chromosome. The better scoring chromosomes are deemed to be more desirable, and hence their data is retained. Conversely, the low scoring chromosomes are discarded from the population and replaced with new chromosomes to form a new *generation*. *Elitism* favors the highest ranking chromosomes, and directly forwards them to the new generation's population. Others are created by recombination of *selected* chromosomes.

The *selection* operator is applied to select chromosomes with likelihood proportional to their fitness score. Different selection schemes exist, including roulette wheel selection and tournament selection. In roulette wheel selection, the probability of selecting the ith chromosome, denoted with b_i, follows a Bernoulli distribution by $p = \phi(b_i)/\sum_{j=0}^{n} \phi(b_j)$.

The *crossover* operator is applied on two randomly selected chromosomes. In canonical GA, a single-point crossover is used, where a position in the binary string is chosen at random and the opposing string sections of the two parents are exchanged, creating two new offsprings.

The *mutation* operator randomly flips single bits on a specific chromosome with a predefined mutation probability. Mutation is necessary to maintain genetic diversity from one generation of a population to the next.

The evolutionary process is repeated until a given termination criterion is satisfied. This criterion may include reaching a predetermined number of generations, finding a chromosome with fitness better than a certain minimum, or lack of improvement in the population fitness despite evolution.

Since its introduction in 1975 (Holland 1975), other techniques and evolutionary algorithms have been proposed to extend canonical GA. For example, to facilitate complex data representation, GA is often implemented with integer or floating point codons and evolutionary operators are applied directly to the codons instead of the underlying bit string. This method also takes the advantage of the architecture of modern processors to speed-up computation. For a review of other GA techniques, readers are referred to a survey by Srinivas and Patnaik (1994).

2.2. Context-free grammar

A context-free grammar (CFG) is a mechanism to generate patterns and strings using hierarchically organized production rules (Sipser 1997). A CFG is described by the tuple $(\mathcal{T}, \mathcal{N}, \mathcal{R}, \mathcal{S})$ where $\mathcal{T}$ is a set of terminal symbols, $\mathcal{N}$ is a set of non-terminal symbols with

$\mathcal{N} = \{expr,\ op,\ coef,\ var\}$
$\mathcal{T} = \{\div,\ \times,\ \texttt{+},\ \texttt{-},\ v_1,\ v_2,\ c_1,\ c_2,\ \texttt{(},\ \texttt{)}\}$
$\mathcal{S} = <expr>$

$\mathcal{R}$ = Production rules:

$\langle expr \rangle$	$::= \ (\langle expr \rangle)\langle op \rangle(\langle expr \rangle)$	(1.a)
	$\mid \langle coef \rangle \times \langle var \rangle$	(1.b)
$\langle op \rangle$	$::= \ \texttt{+} \mid \texttt{-} \mid \times \mid \div$	(2.a), (2.b), (2.c), (2.d)
$\langle coef \rangle$	$::= \ c_1 \mid c_2$	(3.a), (3.b)
$\langle var \rangle$	$::= \ v_1 \mid v_2$	(4.a), (4.b)

Table 1: An example grammar in BNF notation. The three first lines define the non-terminal ($\mathcal{N}$), terminal ($\mathcal{T}$), and start ($\mathcal{S}$) symbol sets respectively. The rest of the lines define the production rules ($\mathcal{R}$).

$\mathcal{N} \cap \mathcal{T} = \varnothing$, and $\mathcal{S} \in \mathcal{N}$ is the *start* symbol. A non-terminal symbol is one that can be replaced by other non-terminal and/or terminal symbols, while terminal symbols are literals. $\mathcal{N}$ and $\mathcal{T}$ form the lexical elements used in $\mathcal{R}$, the production rules of a CFG. $\mathcal{R}$ is defined as a set of relations (also referred to as production rules) in the form of $x \to \alpha$ with $x \in \mathcal{N}$, $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$, where $*$ is the Kleene star. If the grammar rules are defined as $\mathcal{R} = \{x \to xa, x \to ax\}$, a is a terminal symbol since no rule exists to change it.

CFGs are commonly described using Backus-Naur form (BNF, Knuth 1964). To differentiate between terminal and non-terminal symbols in the BNF, the non-terminal symbols are enclosed within angle brackets (i.e., '<' and '>'). Also in each production rule, possible replacement sequences are separated by a vertical bar (i.e., '|').

An example grammar in BNF notation is given in Table 1. In this grammar, the start symbol ($\mathcal{S}$) is <*expr*>. Each of the non-terminal symbols defined in $\mathcal{N}$, <*expr*>, <*op*>, <*coef*> and <*var*>, can be replaced by an appropriate terminal as specified in $\mathcal{R}$. For example, <*expr*> can either expand to (<*expr*>)<*op*>(<*expr*>) or <*coef*>×<*var*>, and <*op*> can be replaced by one of the +, -, ×, or ÷ operators.

2.3. Genotype to phenotype mapping using grammar rules

In evolutionary biology, chromosome data is referred to as the *genotype*, while an organism's observable characteristics are called the *phenotype*. Biological organisms use complicated methods to *map* their genotype to phenotype. Advanced evolutionary algorithms, such as GE, use a similar notion to create complex objects from simple chromosome structures.

In GE, genotype to phenotype mapping is performed according to the production rules of a CFG selected using the chromosome's codon values. The usual mapping function used is the `mod` rule defined as: (codon integer value) `mod` (number of rules for the current non-terminal), where `mod` is the modulus operator. Mapping begins from the start symbol $\mathcal{S}$, and continues by replacing each non-terminal element $\mathcal{N}$ according to the production rule $\mathcal{R}$ chosen by the mapping function. At each step, the resulting expression can contain terminal (i.e., $\mathcal{T}$) or

Step	Codon	`mod` operator	Rule	Current element state
0			$\mathcal{S}$	`<expr>`
1	2	2 `mod` 2	0 (1.a)	`(<expr>)<op>(<expr>)`
2	1	1 `mod` 2	1 (1.b)	`(<coef>`$\times$`<var>)<op>(<expr>)`
3	0	0 `mod` 2	0 (3.a)	$(c_1$ $\times$`<var>)<op>(<expr>)`
4	0	0 `mod` 2	0 (4.a)	$(c_1 \times v_1)$`<op>(<expr>)`
5	3	3 `mod` 4	3 (2.d)	$(c_1 \times v_1) \div$ `(<expr>)`
6	3	3 `mod` 2	1 (1.b)	$(c_1 \times v_1) \div$ `(<coef>`$\times$`<var>)`
7	3	3 `mod` 2	1 (3.a)	$(c_1 \times v_1) \div (c_2$ $\times$`<var>)`
8	1	1 `mod` 2	1 (4.a)	$(c_1 \times v_1) \div (c_2 \times v_2)$

Table 2: Production of an expression using the grammar of Table 1. The process starts from the start symbol $\mathcal{S}$, and continues by replacing the first symbol present in $\mathcal{N}$ with another. This later symbol is selected from the production rules $\mathcal{R}$ according to the value of the current codon. In 8 steps, all of non-terminal symbols are replaced and the string [2|1|0|0|3|3|3|1] is mapped to $(c_1 \times v_1) \div (c_2 \times v_2)$.

non-terminal elements. The mapping continues until all non-terminal elements are replaced with terminals.

If the chromosome is too short, it may run out of codons with non-terminal elements still remaining. A common approach is to *wrap* the chromosome and continue the mapping process by reusing the codons from the beginning. However, in cyclic grammars, infinite recursion may occur. This is addressed by introducing a limit on the number of allowed chromosome wrappings and returning a poor fitness score if the limit is reached.

In this section, the grammar in Table 1 is used as an example of expression generation. Consider the chromosome with a 16-bit genotype, [2|1|0|0|3|3|3|1], where the integer numbers represent 2-bit codon values. There are two production rules to choose from for the start symbol $\mathcal{S}$ = <*expr*>, (1.a) and (1.b). The `mod` operation on the current codon becomes 2 `mod` 2=0, hence rule (1.a) is chosen. The successive application of rules is demonstrated in Table 2, showing how an expression is generated by the example chromosome. The resulting phenotype, $(c_1 \times v_1) \div (c_2 \times v_2)$, can be later evaluated in different contexts as a numerical value.

GE uses the standard evolutionary operators from canonical GAs to evolve the chromosomes and generate new programs. An in-depth explanation of GE can be found in the original GE paper by O'Neill and Ryan (2001).

2.4. Software implementations of GE

Several open source software implementations of GE are available for different programming languages. These include **GEVA** (O'Neill, Hemberg, Gilligan, Bartley, McDermott, and Brabazon 2008) in Java, **PonyGE** (Hemberg and McDermott 2012) and **PyNeurGen** (Smiley 2012) in Python, **GEM** (Hemberg 2011) for MATLAB, and **GERET** (Suchmann 2013) for Ruby. **gramEvol** (Noorian and de Silva 2016) is the first package for R.

The design goal of this package is to evolve programs natively in R. While it is possible to generate and call R code from other languages, a native implementation has the following advantages:

- R's `expression` objects are used to define a grammar, removing an error prone text-based BNF interpretation or compilation step, allowing dynamic grammar manipulation and rapid prototyping.
- Expression are created directly in R as `expression` objects, which removes the overhead of calling R from an external program.
- Only R's base packages are used for evolutionary operations and grammar processing along with parsing and running generated programs. This eliminates the need for third-party libraries and external dependencies.

A disadvantage of **gramEvol** is its speed compared to compiled GE libraries, such as **libGE** (Nicolau 2006) or **AGE** (Nohejl 2011), which are written in C++. We assume that the computational overhead of processing the cost function is greater than the overhead of GE operators. Hence any major speed-up will be a result of moving the cost function computational bottleneck to C, C++ or Fortran. This is already a common practice in the design and implementation of R packages. Furthermore, packages such as **Rcpp** (Eddelbuettel and Francois 2011) are available to facilitate porting existing R code to C++.

3. Package gramEvol

The **gramEvol** package implements grammatical evolution (GE) for R. It offers facilities for defining, creating, evaluating, and evolving programs based on context-free grammars, which are introduced in this section.

3.1. Defining a grammar

In **gramEvol**, a grammar is defined by passing a list of productions rules to the function `CreateGrammar`. `CreateGrammar` automatically determines the terminal, non-terminal and start symbols based on the rules. **gramEvol** supports two type of rules: expression based rules defined using `grule`, and character string rules defined using `gsrule`.

For example, the following commands will construct the CFG of Table 1 using `gsrule`:

```
R> library("gramEvol")
R> ruleDef <- list(expr = gsrule("(<expr>)<op>(<expr>)", "<coef>*<var>"),
+    op   = gsrule("+", "-", "*", "/"), coef = gsrule("c1", "c2"),
+    var  = gsrule("v1", "v2"))
R> grammarDef <- CreateGrammar(ruleDef)
R> grammarDef
```

```
<expr> ::= (<expr>)<op>(<expr>) | <coef>*<var>
<op>   ::= + | - | * | /
<coef> ::= c1 | c2
<var>  ::= v1 | v2
```

Using R's native `expression` objects require a change to the grammar, as `expr op expr` is not valid in R. Instead, a functional form of `op(expr, expr)` is used with `grule`:

```
R> ruleDef <- list(expr = grule(op(expr, expr), coef*var),
+     op   = grule(`+`, `-`, `*`, `/`), coef = grule(c1, c2),
+     var  = grule(v1, v2))
R> CreateGrammar(ruleDef)

<expr> ::= <op>(<expr>, <expr>) | <coef> * <var>
<op>   ::= `+` | `-` | `*` | `/`
<coef> ::= c1 | c2
<var>  ::= v1 | v2
```

The grammar properties are reported via the `summary` function:

```
R> summary(grammarDef)

Start Symbol:                 <expr>
Is Recursive:                 TRUE
Tree Depth:                   Limited to 4
Maximum Rule Choices:         4
Maximum Sequence Length:      18
Maximum Sequence Variation:   2 2 2 2 4 4 2 2 2 4 2 2 2 2 4 2 2 2
No. of Unique Expressions:    18500
```

This summary reports that:

- The non-terminal symbol of the first production rule (i.e., <*expr*>) was selected as the start symbol $\mathcal{S}$.
- The grammar is cyclic, i.e., the non-terminal symbol <*expr*> expands to more <*expr*>s. To avoid infinite recursion, the maximum recursion depth is limited to the number of production rules.
- The grammar tree depth is limited to four.
- Maximum choices in a production rule is four, given by <*op*>.
- Maximum length of a chromosome, avoiding wrapping and limiting recursions, is 18.
- The maximum variation of each integer codon in the chromosome. This value depends on the location of the codons and the grammar, and helps reduce the search space of chromosomes.
- The grammar, with recursion limited to four, can create 18500 different expressions.

`GrammarMap` maps a sequence of integers (the genotype in evolutionary algorithms) to a symbolic expression (the phenotype). The example below converts the numeric genome in Table 2 to its analytical phenotype, using the `verbose` argument to show the steps of the mapping.

```
R> genome <- c(2, 1, 0, 0, 3, 3, 3, 1)
R> expr <- GrammarMap(genome, grammarDef, verbose = TRUE)

 Step Codon Symbol Rule                 Result
 0                 starting:            <expr>
 1    2     <expr> (<expr>)<op>(<expr>) (<expr>)<op>(<expr>)
 2    1     <expr> <coef>*<var>         (<coef>*<var>)<op>(<expr>)
 3    0     <coef> c1                   (c1*<var>)<op>(<expr>)
 4    0     <var>  v1                   (c1*v1)<op>(<expr>)
 5    3     <op>   /                    (c1*v1)/(<expr>)
 6    3     <expr> <coef>*<var>         (c1*v1)/(<coef>*<var>)
 7    3     <coef> c2                   (c1*v1)/(c2*<var>)
 8    1     <var>  v2                   (c1*v1)/(c2*v2)
Valid Expression Found

R> expr

(c1 * v1)/(c2 * v2)
```

The returned object is of class `GEPhenotype`. It can be cast to a character string or an expression and subsequently evaluated using R's `eval` function:

```
R> as.character(expr)

[1] "(c1 * v1)/(c2 * v2)"

R> c1 <- 1
R> c2 <- 2
R> v1 <- 3
R> v2 <- 4
R> eval(as.expression(expr))

[1] 0.375
```

To inspect some random expressions of the grammar, `GrammarRandomExpression` can be used. For the purpose of reproducibility, the random generator seed value is first set to a fixed value:

```
R> set.seed(0)
R> GrammarRandomExpression(grammarDef, numExpr = 4)

[[1]]
expression((c2 * v2) + (c1 * v1))

[[2]]
expression((c1 * v1) - (c1 * v2))
```

```
[[3]]
expression(c1 * v1)

[[4]]
expression((((c1 * v2) - ((c1 * v2) - (c2 * v2))) + ((c1 * v1) +
    (c1 * v2))) - ((c1 * v2) - (c2 * v2)))
```

From the example, it can be seen that this grammar is capable of generating both simple and complex expressions.

3.2. Exhaustive and random search in grammar

Context-free grammars are a general way of describing program structures, not bound to evolutionary optimization. As a result, **gramEvol** additionally supports exhaustive and random search.

The first step in any optimization is defining a cost function. This function receives an expression generated using the grammar, and returns an appropriate score. For example, in order to find the numeric sequence that generates a certain expression, the following cost function returns the generalized Levenshtein distance of the current expression and the target:

```
R> evalFunc <- function(expr) {
+    adist(as.character(expr), "(c1 * v1) - (c2 * v2)")
+  }
```

The objective is to find a suitable chromosome, and therefore the expression, that minimizes the cost function, i.e., the string distance. `GrammaticalExhaustiveSearch` performs an exhaustive search to find this expression:

```
R> GrammaticalExhaustiveSearch(grammarDef, evalFunc)

GE Search Results:
  Expressions Tested: 18500
  Best Chromosome:    0 1 0 0 1 1 1 1
  Best Expression:    (c1 * v1) - (c2 * v2)
  Best Cost:          0
```

`GrammaticalRandomSearch` performs a similar albeit random search. The `terminationCost` option allows the algorithm to terminate if the required minimum cost is found. In our example, the optimal cost is zero:

```
R> GrammaticalRandomSearch(grammarDef, evalFunc, terminationCost = 0)

GE Search Results:
  Expressions Tested: 1000
  Best Chromosome:    0 1 2 0 2 3 1 1 0 3 1 1 0 0 1 1 1 2
  Best Expression:    (c1 * v1) * (c2 * v2)
  Best Cost:          1
```

Both of these methods have their drawbacks: testing 18500 expressions requires extensive computation, and a random search is ineffective. In such cases, considering the non-smoothness and non-convexity of the search space, evolutionary algorithms are often an efficient choice.

3.3. Evolving a grammar

`GrammaticalEvolution` uses evolutionary optimization to find the minima of `evalFunc`. Continuing the previous example, the best expression is determined using the same grammar and cost function, optimized using `GrammaticalEvolution`. Details of evolutionary optimization, such as size of the population and number of iterations are automatically chosen by an internal heuristic:

```
R> result <- GrammaticalEvolution(grammarDef, evalFunc, terminationCost = 0)
R> print(result, show.genome = TRUE)

Grammatical Evolution Search Results:
  No. Generations:  3
  Best Genome:      2 1 0 0 1 1 1 1 0 3 3 1 2 1 2 0 2 1
  Best Expression:  (c1 * v1) - (c2 * v2)
  Best Cost:        0
```

It is evident that the evolutionary algorithm has quickly converged to the optimization objective.

`GrammaticalEvolution` allows monitoring the status of each generation using a callback function. This function, if provided to parameter `monitorFunc`, receives an object similar to the return value of `GrammaticalEvolution`. For example, the following function prints the information about the current generation and the best chromosome in the current generation:

```
R> customMonitorFunc <- function(results) print(results)
R> ge <- GrammaticalEvolution(grammarDef, evalFunc, terminationCost = 0,
+    monitorFunc = customMonitorFunc)
```

Internally, `GrammaticalEvolution` uses `GeneticAlg.int`, which is a GA implementation with integer codons partially based on **genalg** package by Willighagen (2015):

- Using the information obtained about the grammar (e.g., number of possibles expressions and maximum sequence length), `GrammaticalEvolution` applies a heuristic algorithm based on the work of Deb and Agrawal (1999) to automatically determine a suitable value for the `popSize` (i.e., the population size) and the `iterations` (i.e., the number of iterations) parameters.

- The ordinary crossover operator is considered destructive when homologous production rules are not aligned, such as for cyclic grammars (O'Neill, Ryan, Keijzer, and Cattolico 2003). Consequently, `GrammaticalEvolution` automatically changes crossover parameters depending on the grammar to improve optimization results.

- Each integer chromosome is mapped using the grammar, and its fitness is assessed by calling `evalFunc` (i.e., the cost function).

- After reaching a termination criteria, e.g., the maximum number of `iterations` or the desired `terminationCost`, the algorithm stops and returns the best expression found so far.

- `GrammaticalEvolution` also supports multi-gene operations, generating more than one expression per chromosome using the `numExpr` parameter.

`GrammaticalEvolution`'s algorithm is summarized in Figure 3.

```
Function GrammaticalEvolution is
    if missing crossover parameters then
        determine crossover parameters based on grammar
    end
    if missing popSize or iterations then
        determine the optimal popSize and iterations based on grammar
    end
    genotypes ← suggestions
    for i := length(suggestions) + 1 to popSize do
        genotypes $\xleftarrow{Append}$ random chromosome
    end
    for generation := 1 to iterations do
        for i := 1 to len(genotypes) do
            phenotypes[i] ← GrammarMap (grammarDef, genotypes[i], wrappings)
        end
        fitnesses ← evalFunction(phenotypes)
        if terminationCost is given & minimum(fitnesses) < terminationCost
          then
            break For loop
        end
        genotypes ← sort genotypes by their fitness
        new_genotypes ← genotypes[1 to elitism]
        for i := elitism+1 to popSize do
            $parent_1$ ← Select from genotypes using Roulette Wheel operator
            $parent_2$ ← Select from genotypes using Roulette Wheel operator
            new_genotypes[i] ← Crossover($parent_1$, $parent_2$, crossover parameters)
            if random number > mutationChance then
                Mutate new_genotypes[i]
            end
        end
        genotypes ← new_genotypes
    end
    return the genotype and phenotype (i.e., the expression) with the best fitness
end
```

Figure 3: Pseudocode for the GE algorithm implemented in **gramEvol**.

3.4. Parallel processing option

Processing expressions and computing their fitness is often computationally expensive. The **gramEvol** package can utilize parallel processing facilities in R to improve its performance. This is done through the `plapply` argument of `GrammaticalEvolution` function. By default, `lapply` function is used to evaluate all chromosomes in the population.

Multi-core systems simply benefit from using `mclapply` from package **parallel** (R Core Team 2016), which is a drop-in replacement for `lapply` on POSIX compatible systems. The following code optimizes `evalFunc` on 4 cores:

```
R> library("parallel")
R> options(mc.cores = 4)
R> ge <- GrammaticalEvolution(grammarDef, evalFunc, plapply = mclapply)
```

To run **gramEvol** on a cluster, `clusterapply` functions can be used instead. The **gramEvol** package must be first installed on all machines and the evaluation function and its data dependencies exported to all cluster nodes before GE is called. The following example demonstrates a four-process cluster running on the local machine:

```
R> library("parallel")
R> cl <- makeCluster(type = "PSOCK", c("127.0.0.1", "127.0.0.1",
+    "127.0.0.1", "127.0.0.1"))
R> clusterEvalQ(cl, library("gramEvol"))
R> clusterExport(cl, c("evalFunc"))
R> ge <- GrammaticalEvolution(grammarDef, evalFunc,
+    plapply = function(...) parLapply(cl, ...))
R> stopCluster(cl)
```

3.5. Non-terminal expressions

As demonstrated in Section 3.1, a cyclic grammar allows complex expressions to be derived from a compact description. However, if the chromosome is too short, the expression may still contain non-terminal symbols even after wrapping multiple times. For example:

```
R> chromosome <- c(0)
R> expr <- GrammarMap(chromosome, grammarDef)
R> expr

Non-Terminal Sequence:
 ((((<expr>)<op>(<expr>))<op>(<expr>))<op>(<expr>)
```

Non-terminal expressions are identified using `GrammarIsTerminal` function:

```
R> GrammarIsTerminal(expr)

[1] FALSE
```

`GrammaticalEvolution` and other search functions automatically filter non-terminal expressions, and the user does not need to worry about them in practice.

4. Grammatical evolution for machine learning

In this section, three applications of grammatical evolution in statistics and machine learning are explored. Other applications, such as symbolic regression and regular expression discovery using package **rex** (Ushey, Hester, and Krzyzanowski 2016) are explained in the package's vignette.

4.1. Model selection and hyper-parameter optimization

Selecting the best learning model in a machine learning task is often performed in three steps:

- Feature selection, where different features are selected as inputs to for a learning model.
- Model selection, where candidate learning models are compared and one of them is selected.
- Hyper-parameter optimization, where hyper-parameters of the model are optimized for the current objective, (e.g., the kernel type and parameters for kernel methods).

Due to their importance, dedicated packages such as **caret** (Kuhn 2008, 2016) support feature selection and hyper-parameter optimization for many machine learning techniques. Extending these packages to support new algorithms or combining additional steps into their operation, however, require structural changes to the package's code. In this section, we show how CFGs can offer an easily extensible framework for a simultaneous feature selection, model selection and hyper-parameter optimization.

Here, the `ChickWeight` dataset (R Core Team 2016) is used to demonstrate these steps. The objective is to learn the `weight` of a chicken based on the `Time` passed since its birth and its `Diet`. The `Chick` identifier is also included.

We choose a linear model, an artificial neural network (ANN) from **nnet** (Venables and Ripley 2002) and support vector regression (SVR) from **e1071** (Meyer, Dimitriadou, Hornik, Weingessel, and Leisch 2015) as the possible learning algorithms.

```
R> data("ChickWeight")
R> library("e1071")
R> library("nnet")
R> grammarDef <- CreateGrammar(list(
+    learner = grule(function(train.data) {
+      result <- NULL
+      features <-  weight ~ F1 + F2 + F3
+      if (length(attr(terms(features), "variables")) > 2) {
+            capture.output({
+          result <- model
+        })
+          }
+       return(result)
+     }),
+     model   = grule(lm(features, train.data),
+       nnet(features, train.data, size = nn.size),
+       svm(features, train.data, cost = svm.c, svm.hyperparam)),
+     F1 = grule(Time, 0), F2 = grule(Chick, 0), F3 = grule(Diet, 0),
+     nn.size = grule(4, 8, 16),
+     svm.hyperparam = grule(.(kernel = "linear"),
+       .(kernel = "polynomial", degree = svm.degree),
+       .(kernel = "radial", gamma = svm.gamma)),
+     svm.c = grule(0.1, 1, 10, 100, 1000),
+     svm.degree = grule(1, 2, 3, 4, 5),
+     svm.gamma = grule(0.1, 0.2, 0.5, 1.0)))
```

The start symbol, the <*learner*>, has only one production rule, which creates a function that receives the training data and returns the trained model:

- It first selects the appropriate formula of `features`, and if there is at least one regressor variable, it returns a <*model*>. The `features` formula is built by either selecting a variable (i.e., `Time`, `Chick`, and `Diet`), or 0 using <*F1*>, <*F2*>, and <*F3*> rules.
- The <*model*> can be either a `lm`, an `svm` or a `nnet` and is wrapped in `capture.output` to suppress the diagnostic but useless messages by `nnet`.
- Each learning algorithm has its own set of hyper-parameters: `nnet`'s hidden layer `size` is determined using <*nn.size*>, and `svm` uses <*sym.hyperparamm*> to select its kernel and its associated parameter in one-step. Here, `.()` is used to avoid premature interpretation of assignment operator and comma (i.e., `=` and `,`) by R.

The remaining rules, assign certain ranges of values to different hyper-parameters, similar to an ordinary *grid search*.

An example of an expression generated by this grammar is:

```
R> GrammarRandomExpression(grammarDef)

expression(function(train.data) {
    result <- NULL
    features <- weight ~ 0 + 0 + Diet
    if (length(attr(terms(features), "variables")) > 2) {
        capture.output({
            result <- nnet(features, train.data, size = 4)
        })
    }
    return(result)
})
```

This uses `Diet` as a feature, and an ANN with four neurons in its hidden layer as its model.

The grammar can generate 432 unique models:

```
R> summary(grammarDef)

Start Symbol:                <learner>
Is Recursive:                FALSE
Tree Depth:                  4
Maximum Rule Choices:        5
Maximum Sequence Length:     8
Maximum Sequence Variation:  1 2 2 2 3 5 3 5
No. of Unique Expressions:   432
```

To assess each model, a cost function is required. In this example, we define a simple cross-validation test, returning the out-of-sample mean square error (MSE):

```
R> set.seed(0)
R> data("ChickWeight")
R> total.samples <- nrow(ChickWeight)
R> train.ind <- sample(total.samples, trunc(total.samples * 0.8))
R> train.data <- ChickWeight[train.ind,]
R> test.data <- ChickWeight[-train.ind,]
R> eval.chicken <- function(expr) {
+    trainer <- eval(expr)
+    model <- trainer(train.data)
+    if (is.null(model)) {
+      return (Inf)
+    }
+    test.results <- predict(model, test.data)
+    cost <- mean((test.results - test.data$weight)^2)
+    return (cost)
+ }
```

The `eval.chicken` function, first evaluates the expression to get its underlying function. This function is then applied to the training data to obtain a model. If the model is `NULL`, i.e., some error has occurred during the training, it returns a very high cost. Otherwise, the model is used on the testing data, and the MSE of the results is returned.

To find the best combination of features, model and hyper-parameters, Grammatical Evolution is applied to the appropriate grammar and cost function:

```
R> result <- GrammaticalEvolution(grammarDef, eval.chicken)
R> result

Grammatical Evolution Search Results:
  No. Generations:  108
  Best Expression:  function(train.data) {
    result <- NULL
    features <- weight ~ Time + Chick + 0
    if (length(attr(terms(features), "variables")) > 2) {
        capture.output({
            result <- svm(features, train.data, cost = 100,
                          kernel = "radial", gamma = 0.1)
        })
    }
    return(result)
}
  Best Cost:        68.3212558249473
```

The optimal model uses only two of the available features with a radial kernel SVR, and is identical to the result of an exhaustive search:

```
R> GrammaticalExhaustiveSearch(grammarDef, eval.chicken)

GE Search Results:
  Expressions Tested: 432
  Best Chromosome:    0 0 0 1 2 3 2 0
  Best Expression:    function(train.data) {
    result <- NULL
    features <- weight ~ Time + Chick + 0
    if (length(attr(terms(features), "variables")) > 2) {
        capture.output({
            result <- svm(features, train.data, cost = 100,
                          kernel = "radial", gamma = 0.1)
        })
    }
    return(result)
}
  Best Cost:          68.32126
```

To compare the performance of GE and exhaustive search, the GE was run 100 times, with termination condition set to reaching the global optima obtained by the exhaustive search. The error, number of generations and the duration of execution was measured. The tests were performed on a single thread on a 3.40 GHz Intel Core i7-2600 CPU. To ensure reproducibility, `set.seed(0)` was executed before running the code. The results are presented in Table 3. Overall, the GE's average execution time is better than that of the exhaustive search. It must be noted that however, as the GE is an *stochastic* optimization, on some occasions it was unable to find the global minima before reaching the maximum number of allowed iterations. In this example this was limited to 108 generations, set automatically by `GrammaticalEvolution`. As a result, the optimization terminated prior to reaching the global optima.

The final model can be constructed from the results of GE optimization:

```
R> train.func <- eval(result$best$expression)
R> final.model <- train.func(ChickWeight)
```

The machine learning approach used in this section was intentionally kept simple. Other learning algorithms can be added as additional rules, each with their own hyper-parameters.

	Exhaustive search	GE minimum	GE median	GE maximum
Error	68.32	68.32	68.32	988.45
Generations	-	1	39.50	108
Time (s)	71.72	0.61	19.60	149.57

Table 3: Summary of GE's performance for 100 runs of the model selection example.

Different options, such as scaling or dimension reduction techniques can also be added to the <*learner*> function, each described using separate rules.

4.2. Classification

In the second example, we use GE for classification. There are many ways that GE can be adopted for classification, e.g., a model selection on classifiers similar to Section 4.1. Here, we directly define a grammar which takes input variables and returns the classification result, with a structure similar to a decision tree.

In this example, the objective is defined as separating *Iris versicolor* from other species in the Iris flower dataset. Here the data is evaluated from a data-frame instead of the program's environment.

```
R> data("iris")
R> iris$Species <- ifelse(iris$Species == "versicolor", "versicolor", "other")
R> ClassifyFitFunc <- function(expr) {
+    sum(eval(expr, envir = iris) != iris$Species)
+  }
```

The grammar is defined using the following code:

```
R> ruleDef <- list(
+    result = grule(ifelse(expr, "versicolor", "other")),
+    expr = grule((expr) & (sub.expr), (expr) | (sub.expr), sub.expr),
+    sub.expr = grule(comparison(var, func.var)),
+    comparison = grule(`>`, `<`, `==`, `>=`, `<=`),
+    func.var = grule(num, var, func(var)), func = grule(mean, max, min, sd),
+    var = grule(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
+    num = grule(1, 1.5, 2, 2.5, 3, 4, 5))
R> grammarDef <- CreateGrammar(ruleDef)
```

In this grammar, the start symbol, <*result*>, receives a `TRUE`/`FALSE` and returns either `'versicolor'` or `'other'`. The `TRUE`/`FALSE` value is generated by recursively applying boolean operators to <*sub.expr*>s. In turn, each <*sub.expr*> is created by a <*comparison*> of a <*var*> in Iris features and another value created using <*func.var*>.

A few examples of the grammar generated expression, formatted through the `pretty.print` function, are as follows:

```
R> pretty.print <- function(expr) cat(gsub("|", "|\n\t",
+    gsub("&", "&\n\t", as.character(expr), fixed = TRUE), fixed = TRUE),
+    "\n")
```

Value	Minimum	Median	Maximum
Error	4	8	22
Generations	1000	1000	1000
Time (s)	12.56	12.87	13.24

Table 4: Summary of GE's performance for 100 runs of the classification example.

```
R> pretty.print(GrammarRandomExpression(grammarDef))

ifelse(((Petal.Width > Petal.Length) &
         (Sepal.Length >= sd(Petal.Length))) &
         (Petal.Length == 5), "versicolor", "other")

R> pretty.print(GrammarRandomExpression(grammarDef))

ifelse((Sepal.Width == min(Petal.Length)) |
         (Sepal.Length <= sd(Sepal.Length)), "versicolor", "other")
```

The GE optimization is performed by:

```
R> set.seed(10)
R> ge <- GrammaticalEvolution(grammarDef, ClassifyFitFunc)
R> expr <- ge$best$expression
R> pretty.print(expr)

ifelse(((Sepal.Width >= max(Sepal.Length)) |
         (Petal.Width <= sd(Petal.Length))) &
         (Petal.Length >= Sepal.Width), "versicolor", "other")

R> err <- sum(eval(expr, envir = iris) != iris$Species)
R> err

[1] 6
```

The classification results are visualized in Figure 4.

Table 4 summarizes the performance of GE classifier for 100 executions. As no termination condition was given, all of the runs terminated only after reaching the maximum allowed number of generations. It is evident that on average, GE is able to find an acceptable expression with in this limit.

4.3. Symbolic regression and feature generation

Symbolic regression is the process of discovering a function, in analytical form, which fits a given set of data. Commonly, evolutionary algorithms such as GP and GE are used for this task. Symbolic regression suffers from a possibly infinite, non-smooth and non-convex search space, and therefore is not widely used in machine learning.

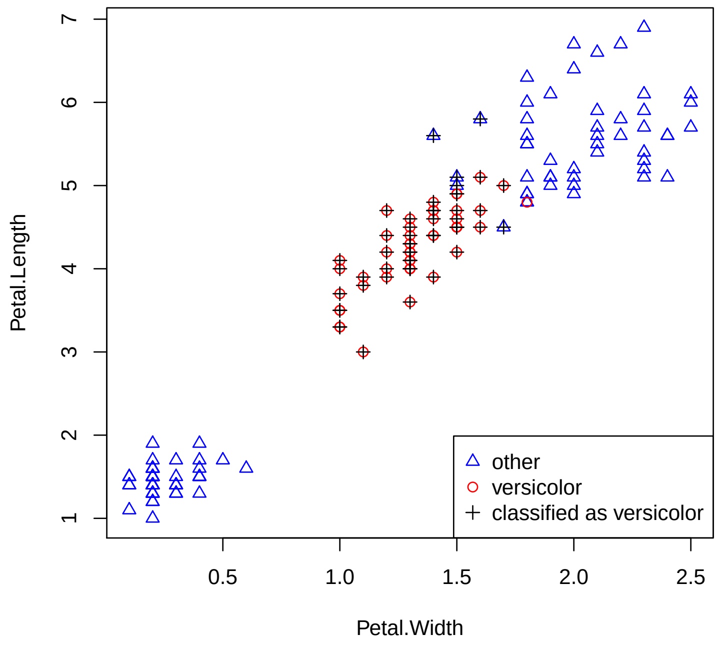

Figure 4: Classification of *Iris versicolor* using GE.

Feature generation is the process of deriving new features from existing features (Guo, Jack, and Nandi 2005). In this technique, an evolutionary algorithm is used to generate and combine results of multiple independently discovered expression, e.g., by using a linear combination of GP results (Keijzer 2004; Costelloe and Ryan 2009), or by using non-linear function estimators applied to GE (de Silva, Noorian, Davis, and Leong 2013). This can be considered a type of machine learning and symbolic regression hybrid, as the final learning model is constructed from combination of simpler *features* created through a process similar to symbolic regression.

For example, consider learning of the following sextic polynomial from numeric data:

$$f(X) = X^6 + X^5 + X^4 + X^3 + X^2 + X + 1$$

Evolving an expression that matches the observed data to this polynomial would either require a very well crafted grammar, or a successful search over a huge space, both of which are extremely computationally expensive.

However, linear dependencies exist between components of this function. By designing a *multi-gene* chromosome, we can generate individual expressions independently and then combine them through a linear regression model to create the final expression. This effectively breaks the search space to several smaller ones, enabling a faster search over the whole space. Figure 5 illustrates the difference between these two approaches.

To compare the symbolic regression and the feature generation with ordinary GE, two approaches are benchmarked using the same grammar:

```
R> ruleDef <- list(expr = grule(op(expr, expr), func(expr), var),
+    func = grule(sin, cos, log, sqrt), op = grule(`+`, `-`, `*`),
+    var = grule(X, X^n, n), n = grule(1, 2, 3, 4))
R> grammarDef <- CreateGrammar(ruleDef)
```

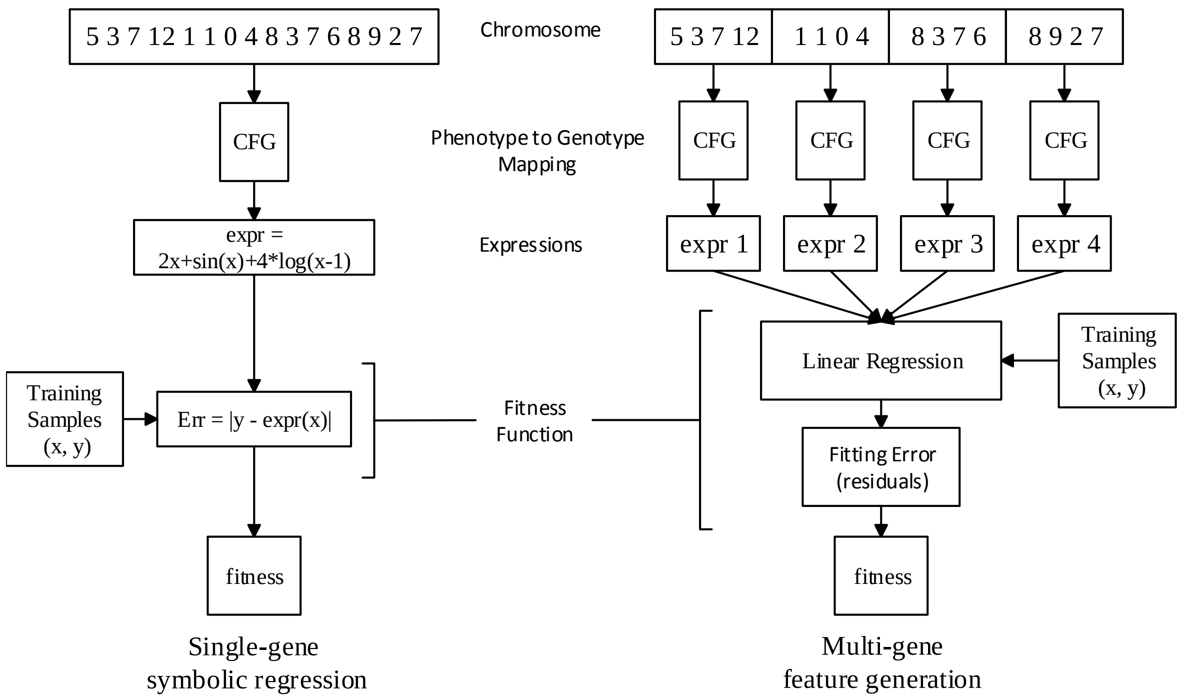

Figure 5: Symbolic regression (using a single gene) vs. feature generation (using multiple genes).

The grammar can be used to generate many different types of expressions:

```
R> set.seed(0)
R> GrammarRandomExpression(grammarDef, numExpr = 3)

[[1]]
expression(log(2))

[[2]]
expression(sqrt(X * X) + cos(sqrt(cos(X^3))) - sqrt(log(sin(X))))

[[3]]
expression(X^3)
```

Obviously, this grammar is not tuned for the purpose of fitting high-degree polynomials.

Symbolic regression

Firstly, symbolic regression is tested:

```
R> target.func <- function(X) X^6 + X^5 + X^4 + X^3 + X^2 + X + 1
R> X <- 1:10
R> Y <- target.func(X)
R> symRegCostFunc <- function(expr) {
```

```
+    result <- suppressWarnings(eval(expr))
+    if (any(is.nan(result)))
+      return (Inf)
+    return (mean((Y - result)^2))
+  }
```

The cost function handles invalid values (e.g., $\log(-1)$) by assigning a high cost to any expression with an invalid value. However, R may show warnings about NaNs being produced. To suppress these warnings, one can wrap the `eval` in the cost function inside `suppressWarnings`.

To allow the GE to have enough room for search, the length of the chromosome is set to 60:

```
R> set.seed(0)
R> ge.single <- GrammaticalEvolution(grammarDef, symRegCostFunc,
+    seqLen = 60, terminationCost = 1e-4)
```

This test is prone to getting stuck in a local minima and multiple restarts may be required to find the solution. Results often lack or include additional terms not in the target, e.g.,

```
R> ge.single

Grammatical Evolution Search Results:
  No. Generations:  1000
  Best Expression:  (X^2 + X^3) * X^3
  Best Cost:        21085073.8
```

The resulting expression can be simplified to $X^6 + X^5$, and therefore has a high error.

Feature generation

The second approach uses `GrammaticalEvolution`'s `numExpr` option to generate multiple expressions. Here, `numExpr = 5` is set, and for a fair comparison, the length allocated to each sequence `seqLen` is also reduced from 60 to 12. `GrammaticalEvolution` will still use a chromosome with length of 60, but this is divided into 5 parts (i.e., the genes), each of which are used individually to generate up to five valid expressions. A simple linear model is then applied to fit these expressions to data and the fitting residuals are reported as error.

For evaluating multiple expression, the function `EvalExpressions` offers a simpler interface compared to `eval`:

```
R> X <- 1:10
R> Y <- target.func(X)
R> fitLinearModel <- function(expr.list) {
+    vals <- EvalExpressions(expr.list)
+    if (any(is.nan(unlist(vals))) | any(is.infinite(unlist(vals))))
+      return(NULL)
+    mdl <- lm(Y ~ ., cbind(as.data.frame(vals), Y = Y))
+    return (mdl)
+  }
```

```
R> fitnessFunction <- function(expr.list) {
+    mdl <- fitLinearModel(expr.list)
+    if (class(mdl) != "lm") return (Inf)
+    return(mean(residuals(mdl)^2))
+  }
```

The `fitnessFunction` uses `fitLinearModel` to create a linear model of generated expressions to data. The model is then fit to the data, and the MSE of residuals are returned as its cost. All other GE parameters (i.e., population size, mutation chance, termination condition, etc.) are kept the same:

```
R> set.seed(10)
R> ge.multi <- GrammaticalEvolution(grammarDef, fitnessFunction,
+    seqLen = 12, numExpr = 5, terminationCost = 1e-4)
```

This approach is clearly better at finding a close approximation to the target:

```
R> ge.multi

Grammatical Evolution Search Results:
  No. Generations:  12
  Best Expressions: X + X^3 * X * ((X + 2) * 1)
                  : X^4 * X^2
                  : X^4
                  : X^2
                  : X^3
  Best Cost:        6.20330039637389e-23

R> expr <- ge.multi$best$expression
R> mdl <- fitLinearModel(expr)
R> mdl

Call:
lm(formula = Y ~ ., data = cbind(as.data.frame(vals), Y = Y))

Coefficients:
(Intercept)        expr1        expr2
          1            1            1
      expr3        expr4        expr5
         -1            1            1

R> X <- seq(1, 10, length.out = 40)
R> pred <- predict(mdl, newdata = EvalExpressions(expr))
R> err <- mean((target.func(X) - pred)^2)
R> err

[1] 4.64618e-21
```

	Symbolic regression			Feature generation		
Value	Minimum	Median	Maximum	Minimum	Median	Maximum
Error	2.14×10^5	9.12×10^8	6.15×10^9	0.00	0.00	1.48
Generations	1000	1000	1000	5	25.5	200
Time (s)	23.24	24.73	25.17	6.59	30.65	309.93

Table 5: Performance of symbolic regression vs. feature generation using GE, compared over 100 runs.

In the results above, all the elements of the sextic equation are found within five expressions. Three of them (X^2, X^3, X^4 and X^6) are found separately, and the other expression, `X + X^3 * X * ((X + 2) * 1)`, contains the linear combination $X + 2X^4 + X^5$. As X^4 is already present separately, the linear regression can extract and combine all elements with the correct y-intercept. Consequently, the regression model $\hat{f}(X)$ perfectly matches the original model:

$$\begin{aligned}\hat{f}(X) &= 1 + (X + X^3 \times X \times (X + 2)) + X^4 \times X^2 - X^4 + X^2 + X^3 \\ &= 1 + X + X^2 + X^3 + X^4 + X^5 + X^6\end{aligned}$$

Comparison

To test the stochastic performance of GE with a single and multiple genes, each method was run 100 times and their error from the target equation was noted. The results are presented in Table 5. The results show major improvements in error, from an average 9.12×10^8 for symbolic regression to a worst cast of 1.48 for the feature generation approach. In comparison, the average time required to process both approaches was almost equal.

5. Conclusion

Context-free grammars provide a concise and versatile mechanism for expressing families of programs. Combined with evolutionary optimization, grammatical evolution creates a powerful framework that allows integration of domain specific knowledge, defined using a grammar, into real-world applications.

The **gramEvol** package allows creation of native R programs using GE. After specifying a grammar and evaluation function, users can employ GE techniques with little additional code. Parallel execution is also supported via parallel computing functions within R.

One disadvantage of GE lies in its stochastic nature, as it does not guarantee the convergence to the global optima. The **gramEvol** package includes an exhaustive search option which can ensure an optimal solution at the expense of computation time.

Acknowledgments

This research was supported by the Australian Research Council's Linkage Projects funding scheme (project number LP110200413); funding from the Faculty of Engineering & Information Technologies, The University of Sydney, under the Faculty Research Cluster Program;

and Westpac Banking Corporation. We also thank Michael Frechtling, Duncan Moss, the associate editor and the anonymous reviewers for their valuable comments and ideas.

References

Costelloe D, Ryan C (2009). "On Improving Generalisation in Genetic Programming." In L Vanneschi, S Gustafson, A Moraglio, I Falco, M Ebner (eds.), *Genetic Programming*, volume 5481 of *Lecture Notes in Computer Science*, pp. 61–72. Springer-Verlag. doi:10.1007/978-3-642-01181-8_6.

de Silva AM, Noorian F, Davis RIA, Leong PHW (2013). "A Hybrid Feature Selection and Generation Algorithm for Electricity Load Prediction Using Grammatical Evolution." In *IEEE 12th International Conference on Machine Learning and Applications ICMLA 2013, Special Session on Machine Learning in Energy Applications*, pp. 211–217. doi:10.1109/icmla.2013.125.

Deb K, Agrawal S (1999). "Understanding Interactions among Genetic Algorithm Parameters." *Foundations of Genetic Algorithms*, pp. 265–286.

Eddelbuettel D, Francois R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Guo H, Jack LB, Nandi AK (2005). "Feature Generation Using Genetic Programming with Application to Fault Classification." *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, **35**(1), 89–99. doi:10.1109/tsmcb.2004.841426.

Hemberg E (2011). *Grammatical Evolution in MATLAB (**GEM**)*. Version 0.2, URL http://ncra.ucd.ie/GEM/.

Hemberg E, McDermott J (2012). ***PonyGE**: A Pony-Sized Implementation of Grammatical Evolution in Python*. Version 0.1.5, URL http://ncra.ucd.ie/Site/GEVA.html.

Holland JH (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence.* The University of Michigan Press, Ann Arbor.

Holland JH (1992). *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge.

Keijzer M (2004). "Scaled Symbolic Regression." *Genetic Programming and Evolvable Machines*, **5**(3), 259–269. doi:10.1023/B:GENP.0000030195.77571.f9.

Knuth DE (1964). "Backus Normal Form vs. Backus Naur Form." *Communications of the ACM*, **7**(12), 735–736. doi:10.1145/355588.365140.

Koza JR (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, volume 1. MIT press.

Kuhn M (2008). "Building Predictive Models in R Using the **caret** Package." *Journal of Statistical Software*, **28**(5), 1–26. doi:10.18637/jss.v028.i05.

Kuhn M (2016). ***caret**: Classification and Regression Training.* R package version 6.0-70, URL https://CRAN.R-project.org/package=caret.

McKay RI, Hoai NX, Whigham PA, Shan Y, O'Neill M (2010). "Grammar-Based Genetic Programming: A Survey." *Genetic Programming and Evolvable Machines*, **11**, 365–396. doi:10.1007/s10710-010-9109-y.

Meyer D, Dimitriadou E, Hornik K, Weingessel A, Leisch F (2015). ***e1071**: Misc Functions of the Department of Statistics (E1071), TU Wien.* R package version 1.6-7, URL https://CRAN.R-project.org/package=e1071.

Mitchell M (1996). *An Introduction to Genetic Algorithms.* MIT Press, Cambridge.

Nicolau M (2006). ***libGE** C++ Library.* Stable Release 0.26, URL http://bds.ul.ie/libGE/.

Nohejl A (2011). ***AGE**: Algorithms for Grammar-Based Evolution.* Version 1.1.1, URL http://nohejl.name/age/.

Noorian F, de Silva AM (2016). ***gramEvol**: Grammatical Evolution for R.* R package version 2.1-3, URL https://CRAN.R-project.org/package=gramEvol.

O'Neill M, Hemberg E, Gilligan C, Bartley E, McDermott J, Brabazon A (2008). "**GEVA**: Grammatical Evolution in Java." *ACM SIGEVOlution*, **3**(2), 17–22. doi:10.1145/1527063.1527066.

O'Neill M, Ryan C (2001). "Grammatical Evolution." *IEEE Transactions on Evolutionary Computation*, **5**(4), 349–358. doi:10.1109/4235.942529.

O'Neill M, Ryan C, Keijzer M, Cattolico M (2003). "Crossover in Grammatical Evolution." *Genetic Programming and Evolvable Machines*, **4**(1), 67–93. doi:10.1023/a:1021877127167.

R Core Team (2016). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sipser M (1997). "Context-Free Grammars." In *Introduction to the Theory of Computation*, chapter 2, pp. 91–122. PWS Publishing Company.

Smiley D (2012). ***PyNeurGen**: Python Neural Genetic Algorithm Hybrids.* Release 0.3, URL http://pyneurgen.sourceforge.net/.

Srinivas M, Patnaik LM (1994). "Genetic Algorithms: A Survey." *Computer*, **27**(6), 17–26. doi:10.1109/2.294849.

Suchmann P (2013). *Grammatical Evolution Ruby Exploratory Toolkit (**GERET**).* URL http://geret.org/.

Ushey K, Hester J, Krzyzanowski R (2016). ***rex**: Friendly Regular Expressions.* R package version 1.1.1, URL https://CRAN.R-project.org/package=rex.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S.* 4th edition. Springer-Verlag, New York.

Willighagen E (2015). ***genalg**: R Based Genetic Algorithm.* R package version 0.2.0, URL https://CRAN.R-project.org/package=genalg.

Affiliation:

Farzad Noorian, Anthony M. de Silva, Philip H. W. Leong
Computer Engineering Lab
School of Electrical and Information Engineering
The University of Sydney
NSW, 2006, Australia
E-mail: farzad.noorian@sydney.edu.au, anthonymihirana.desilva@sydney.edu.au,
philip.leong@sydney.edu.au
URL: http://www.ee.usyd.edu.au/cel/farzad,
http://www.ee.usyd.edu.au/cel/mihirana.desilva,
http://www.ee.usyd.edu.au/people/philip.leong

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
July 2016, Volume 71, Issue 1 *Submitted:* 2014-04-04
doi:10.18637/jss.v071.i01 *Accepted:* 2015-06-03