Rule Induction Using a Reverse Polish Representation

G. F. Davenport
John Innes Centre,
Norwich Research Park
Norwich NR4 7UH
United Kingdom.
guy.davenport@bbsrc.ac.uk

M. D. Ryan
School of Information Systems
University of East Anglia
Norwich NR4 7TJ
United Kingdom.
mdr@sys.uea.ac.uk

V. J. Rayward-Smith
School of Information Systems
University of East Anglia
Norwich NR4 7TJ
United Kingdom.
vjrs@sys.uea.ac.uk

Abstract

It is often necessary to extract simple and understandable rules from databases containing inconsistent records and/or irrelevant fields. In this study we have assessed the feasibility of using a genetic programming (GP) approach to extract a single rule to describe such data. Instead of a tree structure we use a Reverse Polish (post-fix) representation. To assess the performance of the GP algorithm, it is compared to a steepest ascent hill climber algorithm and C5.0, a commercially available data mining algorithm (Quinlan, 1997). On the datasets used, the GP algorithm out-performs both C5.0 and a steepest ascent hill climber in the simplicity and, in most cases, the accuracy of the expressions produced.

1 INTRODUCTION

The advent of the information age has left us with numerous databases containing data collected from virtually any source you could possibly think of. Due to the large size of these databases, analysts need to use automatic techniques to assist them in extracting useful knowledge from this data. Often the analyst is looking for relationships between different fields which can be used to classify new cases. However, many of the fields in the data may not be relevant to the class of the record and as such are not useful as predictors. Furthermore, the data may also be inconsistent because it may contain records that have identical field values and yet are from different classes. The presence of inconsistent records and/or irrelevant fields may obscure any relationships between fields in the data and therefore hinder the discovery of such relationships. The techniques used to extract such knowledge must therefore be able cope with these potential problems as well as present any knowledge found in a form understandable to the user.

The process of extracting extract accurate and applicable rules, either directly from the data (Clark and Niblett, 1989; Holte, 1993) or via the construction decision trees (Breiman *et al.*, 1984; Quinlan, 1986), is called rule induction. The purpose of this study is to assess the feasibility of using a genetic programming (GP) approach to extract a single rule from data containing both inconsistent records (noise) and irrelevant fields. Previous GP approaches to rule induction have either relied on traditional tree-based denotations (Ryan and Rayward-Smith, 1998) or have used a crossover operator that requires the child solutions to be repaired (Freitas, 1997). In this study we employ a Reverse Polish (post-fix) notation which allows the solutions represented as strings, which are easier to manipulate than trees, and permits crossovers to be performed without the need to repair child solutions. For simplicity, Boolean data was used in this study, which allowed the rule to take the form of a Boolean expression that best describes the relationship between the predicted class (output field) and one or more of the remaining (input) fields in the data.

2 A SIMPLE GENETIC PROGRAMMING ALGORITHM FOR EXTRACTING BOOLEAN EXPRESSIONS

We wish to find a single Boolean expression that 'best' describes the relationship between the input fields and the output field in the data. This expression will be in form $\alpha \leftrightarrow \beta$, where α is any combination of the input fields using logical operators and $\beta \in \{0, 1\}$ is the value of the output field. We therefore need some way to represent an expression to support crossover and mutation operations. Since the value of β depends only on the output field we therefore only need to encode the value of α. Using Reverse Polish (RP) notation we can specify α as a logical expression without the need for parentheses to denote the order in which the expression should be evaluated. So, for example, we represent $(f_3 \vee \neg f_1) \wedge f_2$ by $f_3 f_1 \neg \vee f_2 \wedge$. Furthermore, any RP encoded logical expression can therefore be represented by a string of integers, where for example variables (fields) $f_1, f_2, \ldots, f_p$ are represented by the integers $1, 2, \ldots, p$ and logical operators $\neg$, $\vee$ and $\wedge$, are represented by the integers, 0, -1 and -2, respectively. For example the RP expression $f_3 f_1 \neg \vee f_2 \wedge$ would be represented by the integer string 3 1 0 -1 2 -2.

GP operates by producing new (valid) solutions from old solutions via recombination (crossover) and/or mutation. There are two different approaches to implementing the crossover and mutation operators. In the first approach a crossover or mutation would be allowed at any position in the parent solutions(s). However, the child solutions(s) from such an operation may be invalid and if so would need to be repaired before being allowed in the new population. To avoid the necessity of repairing the child solutions, the second approach was used in which a crossover or mutation was only made at point(s) that produced valid solution(s). The method used to determine valid crossover and mutation points was developed from the following method that was also used to evaluate the RP expressions.

Any expression written in RP notation can be evaluated using a stack in following way (Burge, 1964). The expression is read from the left to the right and when a variable is encountered, its value is placed on the top of the stack. When a logical operator is read, one or more values are removed from the top of stack and are used as arguments for the logical operator. The number of values removed from the stack depend on the number of arguments the logical operator required (e.g. one for unary operators and two for binary operators). The value returned by the logical operator is then placed on the stack. Once the whole expression has been read, then the value on the top of the stack is the value of the expression. Hence, the following properties of stack evaluation can be used to define a valid RP encoding.

1. Once the whole expression has been evaluated then there should only be one number left on the stack and this will be the value of the expression.
2. There should always be at least one number on the stack after the first value has been read.

We can therefore define a valid RP expression as follows. First let us define the validation function $v(x_i)$ on a single element in the string x_i which simply returns the net change in the number of values on the stack when this element is read during evaluation. Since our encoding scheme uses negative integers for the binary operators $\vee$ and $\wedge$, 0 for the unary operator $\neg$, and integers greater than 0 for variables then

$$v(x_i) = \begin{cases} -1 & : \quad x_i < 0 \\ 0 & : \quad x_i = 0 \\ +1 & : \quad x_i > 0 \end{cases}$$

The validation function v can now be defined recursively for an integer string of size $z > 1$.

$$v(x_1\ x_2\ \ldots\ x_{z-1}\ x_z) = v(x_1\ x_2\ \ldots\ x_{z-1}) + v(x_z)$$

From the two stack evaluation properties described above we can now define a valid RP encoding of a logical expression, represented by the integers $x_1\ x_2\ \ldots\ x_z$, if and only if

1. $v(x_1\ x_2\ \ldots\ x_z) = 1$ and
2. $v(x_1\ x_2\ \ldots\ x_i) \geq 1\ \forall\ 1 \leq i \leq z$

Using this definition the following theorem was obtained that can be used to determine valid crossover points.

Theorem 1. *Any single-point crossover, represented by the ordered pair (i, j), between valid parents*
$p_a = (a_1\ a_2\ \ldots\ a_i\ a_{i+1}\ \ldots\ a_y)$ *and*
$p_b = (b_1\ b_2\ \ldots\ b_j\ b_{j+1}\ \ldots\ b_z)$
after position i in p_a and j in p_b will produce valid children
$c_a = (a_1\ a_2\ \ldots\ a_i\ b_{j+1}\ \ldots\ b_z)$ *and*
$c_b = (b_1\ b_2\ \ldots\ b_j\ a_{j+1}\ \ldots\ a_y)$
if and only if $v(a_1\ a_2\ \ldots\ a_i) = v(b_1\ b_2\ \ldots\ b_j)$.

Using this theorem two types of crossovers were implemented: a single-point crossover and a two-point crossover. A valid single point crossover (i, j) can be found from the theorem directly, whereas the crossover points $((i, j), (k, l))$ for a two-point crossover were found by choosing two valid single-point crossovers (i, j) and (k, l) such that the following conditions applied.

1. $(i \geq k$ and $j \geq l)$ and
2. $(i, j) \neq (k, l)$

The first condition ensured that together the two crossover pairs produced valid children and the second condition ensured that the crossover actually produced children different from their parents. A two-point crossover could either be constrained or unconstrained. In the constrained crossover an additional condition was imposed on the values of the crossover points $((i,j),(k,l))$ such that

$$v(a_{i+1}\ a_{i+2}\ \dots\ a_k) = v(b_{j+1}\ b_{j+2}\ \dots\ b_l) = 1$$

This condition ensured that the central part of each parent solution exchanged in the crossover formed a valid expression on its own. By constraining the two-point crossover in this way, the crossover is simply exchanging sub-expressions contained within larger expressions. The constrained two-point crossover therefore mimics the crossover normally used in GP (Cramer, 1985; Koza, 1992), where programs (solutions) are normally represented by decision trees and the crossover operator is performed by exchanging selected sub-trees between parent programs. In contrast, the unconstrained two-point crossover allows the interchange of any elements between the parents, provided only that the first two conditions are obeyed.

Four types of mutation were implemented: 'point', 'insertion', 'deletion' and 'all mutations'. In all four types, a position i within the solution was chosen at random and then one of following actions was carried out, according to the type of the mutation.

Point mutation If the element at position i represents a field then change its value to represent a different field. Otherwise if the element at position i represents a $\wedge$ or a $\vee$ then change its value to represent a $\vee$ or a $\wedge$, respectively.

Insert mutation At random choose a logical operator $\in \{\wedge, \vee, \neg\}$ and if the operator is a $\neg$ then insert a new element to represent a $\neg$ into the solution at position i. Otherwise, choose a field at random and insert two new elements at position i to represent the field and operator, respectively.

Deletion mutation If the element at position i represents a $\neg$ remove it. If the element at position i represents a $\wedge$ or a $\vee$ then remove it and the nearest element to the left that represents a field. If the element at position i represents a field then remove it and the nearest element to the right that represents either a $\wedge$ or a $\vee$.

All mutations One of the three mutations described above was chosen at random and used.

Both the point and insert mutations will always produce a valid solution. However, the deletion mutation may produce a solution with one or more $\neg$ operators at the beginning of the solution string, which would invalidate the solution. When this occurred, the string element(s) representing the $\neg$ operator was removed from the beginning of the solution string. The insert mutation was also used to create valid solutions for the initial population according to the following protocol. First, start with a integer string containing a single element representing any field chosen at random. Next, use the insert mutation to introduce 0-20 element pairs at random into this string to a valid solution.

The objective of the GP is to find a simple Boolean expression that describes the relationship between the input fields and the output fields of a dataset. As we are only concerned with finding a relationship within the given data and not with finding a rule to classify new cases, the performance of the expression was assessed directly from its error rate on the given (training) data. We therefore want to minimise both the error rate and the length of the expression. Hence, the GP algorithm was set to minimise the following fitness function.

$$fitness = r + \lambda l \qquad (2.1)$$

where r is the error rate of the expression (expressed as a percentage), l is the length in string elements required to encode the expression and $\lambda \geq 0$ is a constant that determines the relative importance of the error rate and the simplicity of the expression. In this study the value of λ was set to 0.01 to ensure that preference was given to the accuracy of the expression.

The basic GP algorithm used for the analysis was developed as part of a package of modern heuristic techniques called the Templar framework (Jones, 1998) and is based on the simple genetic algorithm described by Goldberg (1989). Solutions were ranked according to their fitness and selected with probability proportional to this rank. Each pair of selected solutions was subjected to a crossover operator and then a mutation operator, with probability defined by the crossover and mutation rates, respectively. Child solutions were placed in a separate population, which replaced the orginal population at the end of the generation. In addition the best solution was always maintained in the population from one generation to the next. Each analysis consisted recording the average best solution from for 5 or 10 separate runs. A population size of 500 solutions was used and the algorithm was stopped after 50 generations in which there was no improvement in the best fitness. For each analysis crossover and mutation rates of 0.3 and 0.8, respectively were used since these values gave the best results initial trials of the algorithm on the training data.

3 SEARCHING FOR SIMPLE BOOLEAN EXPRESSIONS

In order to assess the ability of the GP algorithm to extract simple Boolean expressions from Boolean data, four training datasets were constructed that contained a single output field and three, four, five or six input fields. The value of each input field was chosen at random, with equal probability given to either T or F. The relationship between the input fields ($I1$, $I2$, $\ldots$ $I6$) and the output field (O) was determined by one of four Boolean expression described in Table 1. Each of the datasets contained 100 records and was named according to the expression they describe. To assess the ability of GP to cope with irrelevant fields, the algorithm was run on datasets that contained 0 to 100 random fields in addition to the input fields.

To determine how the presence of inconsistent records in the data affected the performance of the different algorithms, noise was introduced into the output fields of the datasets. This noise consisted of flipping the output value in a fixed number of records chosen at random from a dataset. In this way datasets containing 5% to 20% inconsistent (noisy) records were constructed and the algorithm was run on each dataset. The performance of the algorithm using different genetic operators was evaluated by comparing its relative performance using the two different two-point crossovers (constrained and unconstrained) and two different mutation operators (point mutation and all mutations).

The results of the analysis for the 6fields data are presented in Figure 1 and suggest that the 'all mutations' operator is more effective than the point mutation. Hence the 'all mutations' operator was preferred in the next analysis. The results also suggest that there is little difference in performance of the algorithm when using the unconstrained or the constrained crossover on the 6fields datasets, containing no noise. However, when the constrained crossover is used, the algorithm takes noticeably longer to run than when using the unconstrained crossover and yet a similar number of generations are required. This finding suggests that constraining the crossover does not affect the quality of the solutions produced, but does increase the time it takes to reach these solutions. For this reason the unconstrained crossover was preferred over the constrained crossover in the next analysis.

The performance of GP on the 3fields, 4fields, 5fields and 6fields datasets described above was compared to that of the C5.0 algorithm, a commercial data mining algorithm. C5.0 produced two sets of rules, one for each of the output values, 0 and 1. Individual rules that predict the same output value were combined by conjunction to form a single rule for each output value. Finally, these rules were also combined to form a single expression to cover all of the records in the dataset, which can then be compared directly with the expressions produced by GP. The expressions produced in this way were often complex but could be reduced to simpler expressions. However, the processor time required for the GP algorithm was 10-100 times longer than that required to run the C5.0 algorithm.

As an additional control, a steepest ascent hill climber (HC) algorithm was also run on the same datasets. This hill climber starts with a random solution created using the same method used by the GP algorithm and tries to improve this solution by trying all possible neighbourhood move operators on each position in the solution. These move operators were based on the point, insertion and deletion mutation operators used for GP. Since each run of the GP algorithm started with a population of 500 random solutions, the HC was run 500 times on each dataset and the best solution was chosen. The results from this analysis are presented in Table 2.

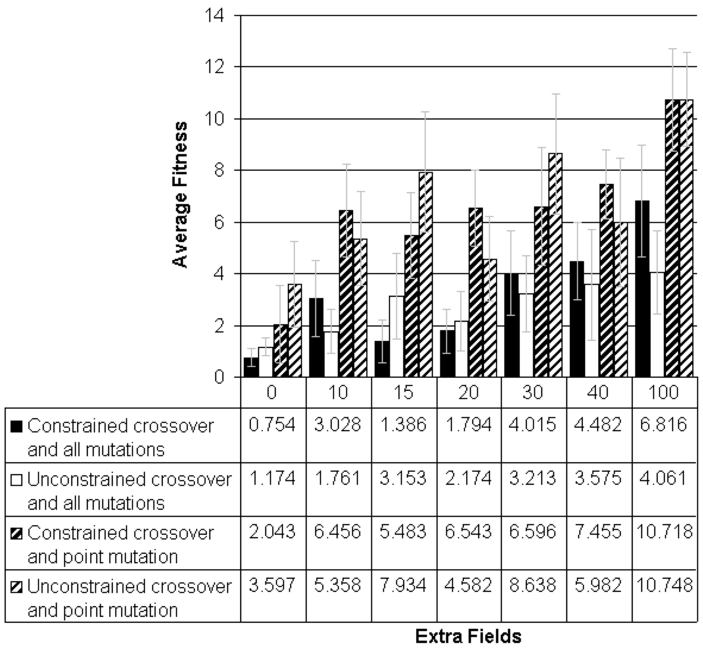
4 DISCUSSION AND FUTURE DIRECTION

We have previously demonstrated that GP can be applied effectively to evolve decision trees (Ryan and Rayward-Smith, 1998) using an algorithm that evolved new decision trees by exchanging subtrees between the trees in the population. Furthermore, GP has also been used to evolve Boolean expressions using tree based denotations (discussed in Koza, 1992). However, the aim of this study was to determine the feasibility of using GP with a Reverse Polish representation, instead of a tree structure, to perform rule induction. In the initial experiments described in this study the comparison between the unconstrained and constrained two point crossovers showed that using the constrained crossover did not improve the performance of the algorithm but did decrease the average execution time. Since, the constrained crossover is equivalent to the crossover used for the tree representation this finding suggests that the Reverse Polish representation is more versatile than trees.

To assess the performance of the GP algorithm on data containing inconsistent records and/or irrelevant fields it was compared with the C5.0 algorithm and a simple HC algorithm. The results presented in Table 2 show that GP outperformed the HC on all of

Table 1: Boolean expressions used in this study

Name	**Expression**
3fields	$(I1 \wedge I2) \vee I3 \leftrightarrow O$
4fields	$((I1 \wedge I2) \vee I3) \wedge \neg I4 \leftrightarrow O$
5fields	$((I1 \wedge I2) \vee I3) \wedge (\neg I4 \vee I5) \leftrightarrow O$
6fields	$(((I1 \wedge (I2 \wedge I6)) \vee I3) \wedge (\neg I4 \vee I5) \leftrightarrow O$



	0	10	15	20	30	40	100
■ Constrained crossover and all mutations	0.754	3.028	1.386	1.794	4.015	4.482	6.816
□ Unconstrained crossover and all mutations	1.174	1.761	3.153	2.174	3.213	3.575	4.061
▨ Constrained crossover and point mutation	2.043	6.456	5.483	6.543	6.596	7.455	10.718
▨ Unconstrained crossover and point mutation	3.597	5.358	7.934	4.582	8.638	5.982	10.748

Figure 1: Analysis of the effect of different crossover and mutation types on the performance of the GP algorithm run on the 6fields data. The average fitness is the mean fitness of the best solutions from 5 separate runs and error bars denote one standard deviation from the mean.

Table 2: Comparison of GP, C5.0 and Hill-climber (HC) techniques on data containing 0%, 5% or 10% noise. The values indicate the number of additional fields that can be added, up to a maximum of 100, before the algorithm was unable to extract the embedded expression. An x denotes that the embedded expression was not found on these datasets.

	noise	**3fields**	**4fields**	**5fields**	**6fields**
GP	0%	100	100	100	10
C5.0	0%	100	100	100	100
HC	0%	100	40	100	x
GP	5%	100	100	0	0
C5.0	5%	100	100	x	x
HC	5%	0	0	x	x
GP	10%	0	20	0	0
C5.0	10%	x	5	x	x
HC	10%	0	0	x	x

the datasets analysed (containing $\leq 10\%$ noise), suggesting that the GP approach was more effective than a greedy local search approach. With the exception of the 6fields datasets, GP performed as well as the C5.0 algorithm on data with no inconsistent fields (0% noise). However, when 5% and 10% noise was introduced into the datasets, GP performed equally or better than the C5.0 algorithm. Furthermore, in contrast to the expressions produced by the C5.0 algorithm, the GP approach produced simple expressions that did not require further simplification. With 20% noise the embedded Boolean expression provides an upper bound on the fitness of the optimal solution. In our experiments the GP algorithm was the only algorithm to produce solutions which were as good or better than this bound. This result shows that with increased noise GP is easily the best option.

The results presented in this study suggest that the GP can extract simpler and in some cases more accurate expressions than the C5.0 algorithm in noisy data. However, the GP algorithm requires much more processor time to produce an expression than that required by C5.0. This finding is not surprising since for every new expression produced by GP its fitness must be calculated by evaluating its performance on each record of the dataset. Furthermore, GP also requires significant processor time to find valid crossover points before a crossover can be performed. Hence, reducing the size of the population or the crossover rate will decrease the time taken by GP to perform a generation. Preliminary studies, not reported in this study, suggest that reducing the size of the population or the crossover rate produced poorer solutions. We therefore need to improve the efficiency of the evaluation and crossover procedures.

The performance of many of the solutions in a population is likely to be poor, particularly at the start of the algorithm. The speed of the evaluation procedure could therefore be increased by determining the fitness of a solution from only a proportion of the records in the database at first. Only when the solution's fitness is better than a defined threshold fitness, are the remaining records used to produce a more accurate measure of the solution's fitness. Using this method less processor time would be spent evaluating the fitness of poor solutions. Since, the time taken to evaluate fitness and to find valid crossover points is dependent on the length of the solutions involved, a further improvement in the speed of the algorithm could be made by limiting the size of the solutions. The size of a solution could be limited either by penalising the fitness of long solutions or by using a pruning procedure to reduce the length of long solutions.

References

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA., 1984.

W. H. Burge. Interpretation, stacks and evaluation. In P. Wegner, editor, *Introduction to Systems Programming.* Academic Press, 1964.

P. C. Clark and T. N. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4), 1989.

N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 183–187, 1985.

A. A. Freitas. A genetic programming framework for two data mining tasks: Classification and generalized rule induction. In John R. Koza *et al.* , editor, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 96–101, Stanford University, CA, USA, 1997. Morgan Kaufmann.

D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, MA, 1989.

R. C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11, 1993.

M. Jones. The templar framwork. *School of Information Systems Technical Report SYS-C98-01, University of East Anglia, UK*, 1998.

J. R. Koza. *Genetic Programming: On the programming of computers by measn of natural selection.* MIT press, Cambridge, MA, 1992.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

J. R. Quinlan. Successor to c4.5. *Knowledge Discovery Nuggets*, 97(09), 1997.

M. D. Ryan and V. J. Rayward-Smith. The evolution of decision trees. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 350–358. Morgan Kaufmann, 1998.