Available online at www.sciencedirect.com
ScienceDirect
Expert Systems with Applications 35 (2008) 1513–1532
Expert Systems with Applications
www.elsevier.com/locate/eswa

# Evolving model trees for mining data sets with continuous-valued classes

Gavin Potgieter, Andries P. Engelbrecht *

*Department of Computer Science, University of Pretoria, South Africa*

**Abstract**

This paper presents a genetic programming (GP) approach to extract symbolic rules from data sets with continuous-valued classes, called GPMCC. The GPMCC makes use of a genetic algorithm (GA) to evolve multi-variate non-linear models [Potgieter, G., & Engelbrecht, A. (2007). Genetic algorithms for the structural optimisation of learned polynomial expressions. *Applied Mathematics and Computation*] at the terminal nodes of the GP. Several mechanisms have been developed to optimise the GP, including a fragment pool of candidate non-linear models, *k*-means clustering of the training data to facilitate the use of stratified sampling methods, and specialized mutation and crossover operators to evolve structurally optimal and accurate models. It is shown that the GPMCC is insensitive to control parameter values. Experimental results show that the accuracy of the GPMCC is comparable to that of NeuroLinear and Cubist, while producing significantly less rules with less complex antecedents.


*Keywords:* Data mining; Continuous-valued classes; Genetic programming; Model trees

## 1. Introduction

*Knowledge discovery* is the process of obtaining useful knowledge from raw data or facts. Knowledge can be inferred from data by a computer using a variety of *machine learning* paradigms. Data mining is the generic term given to knowledge discovery paradigms that attempt to infer knowledge in the form of rules from structured data using machine learning.

Knowledge discovery algorithms can be divided into two main categories according to their learning strategies:

- **Supervised** learning algorithms attempt to minimise the error between their predicted outputs and the target outputs of a given dataset. The target outputs can either be
  - *discrete*, *i.e.* the supervised learning algorithm attempts to predict the class of a problem, *e.g.* whether it will be sunny, rainy or overcast in tomorrow's forecast,
  - or *continuous*, *i.e.* the supervised learning algorithm attempts to predict the value associated with a class, *e.g.* determining the price of a VCR.
- **Unsupervised** learning algorithms attempt to cluster a dataset into homogeneous regions, according to some characteristic present in the data.

Many knowledge discovery algorithms have been developed which utilise machine learning and artificial intelligence paradigms. The main classes of paradigms include: artificial neural networks (Bishop, 1992; Zurada, 1992), classification systems (like ID3 (Quinlan, 1983), CN2 (Clark & Boswell, 1991; Clark & Niblitt, 1989) and C4.5 (Quinlan, 1993)), evolutionary computation (Bäck, Fogel, & Michalewicz, 2000a, 2000b; Goldberg & Deb, 1991), and regression systems (like M5 Quinlan, 1992).

One of the primary problems with current data mining algorithms is the scaling of these algorithms for use on large databases. Additionally, very little attention has been paid to algorithms for mining with continuous target outputs, which requires non-linear regression.

* Corresponding author. Tel.: +27 12 420 3578; fax: +27 12 362 5188.
*E-mail address:* engel@cs.up.ac.za (A.P. Engelbrecht).

0957-4174/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2007.08.060

This paper presents a new approach to extract symbolic rules from data sets where the consequents of the extracted rules are non-linear models. The rule extraction process utilises a genetic programming (GP) approach to evolve model trees. In order to reduce computational effort, to increase the accuracy of rules, and to reduce the complexity of rules, a number of algorithms have been developed to optimise the learning process. These algorithms include:

- specialized mutation and crossover operators to optimise rule sets,
- a fast clustering algorithm (Potgieter & Engelbrecht, 2007) to facilitate stratified sampling of the training data in order to reduce training time, and
- a fragment pool of candidate non-linear models to evolve structurally optimal and accurate models as rule consequents.

In addition to these algorithms, a genetic algorithm (GA) is used to evolve structurally optimal polynomials. These polynomials are used as the models in the fragment pool. More detail on this GA can be found in Potgieter and Engelbrecht (2002, 2007).

The remainder of this paper is organised as follows: Section 2 provides a background to techniques that do not implement greedy search algorithms to generate rules. The structure and implementation specifics of the GPMCC method are discussed in Section 3. Section 4 presents the experimental findings of the GPMCC method for a number of real-world and artificial databases. Finally, Section 5 presents the summarised findings and envisioned future developments to the GPMCC method.

## 2. Background on mining continuous classes

Knowledge discovery algorithms like C4.5 (Quinlan, 1993 and M5 Quinlan, 1992) utilise metrics based on information theory to partition the problem domain and to generate rules. However, these algorithms implement a *greedy* search algorithm to partition the problem domain. For a given attribute space, C4.5 and M5 attempt to select a test that minimises the relative entropy of the subsets resulting from the split. This process is applied recursively until all subsets are homogeneous or some accuracy threshold is reached. Due to their greedy nature, C4.5 and M5 may not generate the smallest possible number of rules for a given problem (Potgieter, 2003). A large number of rules results in decreased comprehensibility, which violates one of the prime objectives of data mining.

This paper discusses a regression technique that does not implement a greedy search algorithm. The regression technique utilises a genetic program for the mining of continuous-valued classes (GPMCC) which is suitable for mining large databases. Although the majority of continuous data is linear, there are cases for which a non-linear approximation technique could be useful, *e.g.* time-series. Therefore, the GPMCC method utilises the GA method presented in Potgieter and Engelbrecht (2002, 2007) to provide non-linear approximations (models) to be used as the leaf nodes (terminal symbols) of a model tree.

This section presents a detailed discussion of two existing methods suitable for non-linear regression. A novel method of generating comprehensible regression rules from a trained artificial neural network is discussed in Section 2.1. Finally, Section 2.2 presents genetic programming approaches for non-linear regression and model tree induction.

### 2.1. Artificial neural networks

Artificial neural networks (ANNs) are widely used as a tool for solving regression problems. However, ANNs have one critical drawback: the complex input to output mapping of the ANN is impossible for a human user to easily comprehend. ANNs are thus one of a handful of *black-box* methods that do not satisfy the comprehensibility requirement of knowledge discovery. While much research has been done on the extraction of rules from ANNs (Fu, 1994; Towell & Shavlik, 1994), these efforts concentrated on classification problems where the target class has discrete values. Not much research has been done for target classes with continuous values. Such problems are usually handled by discretizing the continuous-valued targets to produce a set of discrete values. This section discusses recent work by Setiono that allows decision rules to be generated for regression problems from a trained ANN, called NeuroLinear (Setiono, 2001, Setiono, Leow, & Zurada, 2002). Setiono's findings indicated that rules extracted from ANNs were more accurate than those extracted by various discretisation methods. The section is divided into three parts. Section 2.1.1 discusses the training and pruning strategy of the ANN used by Setiono. Section 2.1.2 describes how a piecewise linear approximation of the activation function is obtained, and Section 2.1.3 discusses the algorithm for generating rules from a trained ANN.

#### 2.1.1. Network training and pruning

The method starts by training an ANN that utilises hyperbolic tangent activation functions in the hidden layer (of size $H$). Training is performed on a training set of $|P|$ training points $(\mathbf{x}_i, y_i)$, $i = 1, \ldots, |P|$ where $\mathbf{x}_i \in \Re^N$ and $y_i \in \Re$. Training, in this case, minimises the sum of squares error $E_{SS}(\mathbf{w}, \mathbf{v})$ augmented with a penalty term $P(\mathbf{w}, \mathbf{v})$.

$$E_{SS}(\mathbf{w}, \mathbf{v}) = \sum_{i=1}^{|P|} (y_i - y_i^{\bullet})^2 + P(\mathbf{w}, \mathbf{v}) \tag{1}$$

with

$$P(\mathbf{w}, \mathbf{v}) = \epsilon_1 \left( \sum_{m=1}^{H} \left( \sum_{l=1}^{N} \frac{\eta w_{ml}^2}{1 + \eta w_{ml}^2} + \frac{\eta v_m^2}{1 + \eta v_m^2} \right) \right) + \epsilon_2 \left( \sum_{m=1}^{H} \left( \sum_{l=1}^{N} w_{ml}^2 + v_m^2 \right) \right) \tag{2}$$

where $\epsilon_1$, $\epsilon_2$ and $\eta$ are positive penalty terms, and $y_i^\bullet$ is the predicted output for input sample $\mathbf{x}_i$, *i.e.*

$$y_i^\bullet = \sum_{m=1}^{H} \tanh((\mathbf{x}_i)^{\mathrm{T}} v_m) + \tau, \tag{3}$$

$\mathbf{w}_m \in \Re^N$ is the vector of network weights from the input units to hidden unit $m$, $w_{ml}$ is the $l$th component of $\mathbf{w}_m$, $v_m \in \Re$ is the network weight from the hidden unit $m$ to the output unit and $\tau$ is the output unit's bias.

Setiono performed training using the BFGS optimisation algorithm, due to its faster convergence than gradient descent (Dennis & Schnabel, 1983; Setiono & Hui, 1995). After training, irrelevant and redundant neurons were removed from the ANN using the N2PFA (Neural Network Pruning for Function Approximation) algorithm (Setiono & Leow, 2000). ANN pruning prevents the ANN from overfitting the training data, and also reduces the length of the rules extracted from the ANN (Viktor, Engelbrecht, & Cloete, 1995). The length of the extracted rules are reduced because the number of variables (weights, input units and hidden units) affecting the outcome of the ANN are reduced.

### 2.1.2. Activation function approximation

The complex input to output mapping of an ANN is a direct consequence of using either the hyperbolic tangent or the sigmoid function as artificial neuron activation functions. In order to generate comprehensible rules, a 3-piece linear approximation of the hyperbolic tangent activation function is constructed. This entails finding the cut-off points ($net_0$ and $-net_0$), the slope ($\beta_0$ and $\beta_1$) and the intersection ($0, \alpha_1$ and $-\alpha_1$) of each of the three line segments. The sum squared error between the 3-piece linear approximation and the activation function is minimised to obtain the values for each of these parameters:

$$\min_{net_0,\beta_0,\beta_1,\alpha_1} = \sum_{i=1}^{|P|} (\tanh(net_i) - L(net_i))^2$$

where $net_i = \mathbf{x}_i^{\mathrm{T}} \cdot \mathbf{w}$ is the weighted input of sample $i$ and

$$L(net) = \begin{cases} -\alpha_1 + \beta_1 net & \text{if } net < -net_0 \\ \beta_0 net & \text{if } -net_0 \leqslant net \leqslant net_0 \\ \alpha_1 + \beta_1 net & \text{if } net > net_0 \end{cases}$$

The intercept and slopes which minimises the sum squared error are calculated as follows:

$$\beta_0 = \frac{\sum_{|net_i| \leqslant net_0} net_i \tanh(net_i)}{\sum_{|net_i| \leqslant net_0} net_i^2}$$

$$\beta_1 = \frac{\sum_{|net_i| > net_0} (net_i - net_0)(\tanh(net_i) - \tanh(net_0))}{\sum_{|net_i| > net_0} (net_i - net_0)^2}$$

$$\alpha_1 = (\beta_0 - \beta_1) net_0$$

The weighted input $net_i$ of each sample is checked as a possible optimal value for $net_0$ starting from the one that has the smallest magnitude. Fig. 1 illustrates how the 3-piece linear approximation is constructed.

### 2.1.3. Generation of regression rules

Linear regression rules are generated from a pruned ANN once the network hidden unit activation functions tanh(*net*) has been approximated by the 3-piece linear function described above. The regression rules are generated as follows:

1. For each hidden unit $m = 1, \ldots, H$
   (a) Generate a 3-piece linear approximation $L_m(net)$.
   (b) Using the points $-net_{m0}$ and $net_{m0}$ from function $L_m(net)$, divide the input space into 3 subregions.
2. For each non-empty subregion $r = 1, \ldots, 3^H$, generate a rule as follows:

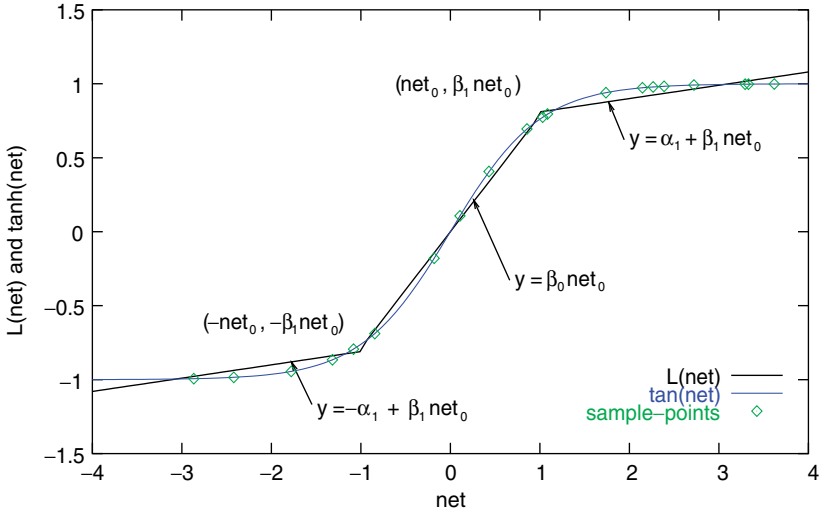


Fig. 1. A 3-piece linear approximation of the hidden unit activation function tanh(net) given 20 training samples (◊).

(a) Define a linear equation that approximates the ANN's output for an input pattern $i$ in subregion $r$ as the rule *consequent*:

$$y_i^{\bullet} = \sum_{m=1}^{H} v_m L_m(net_{mi}) + \tau$$

where $net_{mi} = \mathbf{x}_i^{\mathrm{T}}\mathbf{w}_m, v_m \in \Re$ is the network weight from the hidden unit $m$ to the output unit and $\tau$ is the output unit's bias.

(b) Generate the rule *antecedent*: $((C_1) \wedge \cdots \wedge (C_m) \wedge \cdots \wedge (C_H))$ where $C_m$ is either $net_{mi} < -net_{m0}$, $net_{mi} > net_{m0}$ or $-net_{m0} < net_{mi} < net_{m0}$. Each $C_m$ represents an attribute test. The antecedent is formed by the conjunction of the appropriate tests from each of the hidden units.

(c) Simplify the rule antecedent.

The rule antecedent $((C_1) \wedge \cdots \wedge (C_m) \wedge \cdots \wedge (C_H))$ defines the intersection of each subspace in the input space. For a large number of hidden neurons, this antecedent becomes large. If this antecedent is simplified, using formal logic or by using an algorithm such as C4.5, then the rules generated by the above algorithm will be much easier to comprehend.

## 2.2. Genetic programming

The evolutionary computing paradigm of genetic programming (GP) (Koza, 1992; Poli & Langdon, 2002) can be used to solve a wide variety of data mining problems. This section discusses GP methods for symbolic regression, decision-, regression- and model tree induction, and scaling problems associated with GP. GP satisfies the comprehensibility requirement of knowledge discovery, because the representation of an individual (or chromosome) can be engineered to provide easily understandable results.

### 2.2.1. Symbolic regression

Unlike artificial neural networks, GP can be used to perform *symbolic regression* without the need for data transformations. GP is also capable of regression analysis on variables that exhibit non-linear relationships, as apposed to linear regression techniques. GP is thus a useful tool for regression analysis of non-linear data.

Regression problems are solved by fitting a function, represented by a chromosome, to the dataset using a fitness function that minimises the error between them. The terminal set is defined as a number of constants and attributes, *e.g.* $\{32, 2.5, 0.833, x, y, z\}$, and describes a number of valid states for the leaf nodes of a chromosome (in the form of a tree). The function set is defined by a number of operators, *e.g.* $\{+, -, *, \backslash, \cos, \sin\}$, and describes a number of valid states for the internal nodes of a chromosome.

The constants used in the terminal set are an Achilles' Heel of a symbolic regression genetic program. If a population is liberally scattered with constants chosen from a preset range, *e.g.* $[-1, 1]$, it may be difficult for a genetic program to evolve the expression $300x$. Abass et al. present a concise overview of methods to correct this problem (Abass, Saker, & Newton, 2002).

### 2.2.2. Decision-, regression- and model tree induction

A number of different classification systems that utilise genetic programming (GP) have been developed. This section discusses two interesting ones. Additionally, this section shows how a genetic program can be developed to directly evolve decision-, regression- and model trees.

Eggermont et al. present a GP approach that utilises a *stepwise adaptation of weights* (SAW) technique in order to learn the optimal penalisation factor for the GP fitness function (Eggermont, Eiben, & van Hemert, 1999). Each individual in the population represents a classification rule, and utilises a function set of boolean connectives and a terminal set of attribute tests, *i.e.* either the rule condition covers a training pattern, in which case it is asserted to belong to a class, or it does not. The approach was shown to have increased accuracy over the fixed penalisation factor case.

Freitas presents an interesting GP framework that evolves SQL queries in order to increase scalability, security and portability (Freitas, 1997). Each individual consists of two parts: a tuple-set descriptor and a goal attribute. The GP approach utilises a niching strategy in order to force the method to produce innovative rules.

GP can also be used to directly build decision-, regression- and model trees. As was mentioned in Section 2.1, artificial neural networks provide no comprehensible explanation of how they classify or approximate a dataset. On the other hand, classification systems, such as C4.5, and regression systems, such as M5, generate overly complex trees. GP is potentially capable of providing a compromise between these two extremes.

For zero-order learning, the function set of the chromosome consists of a number of attribute tests. The terminal set for the chromosome consists of either a number of **classes** for decision trees, or a number **values** for regression trees, or a number of **models** for model trees. The models for model trees can be obtained by linear regression, symbolic regression or a population-based optimisation method such as a GA (Potgieter, 2003; Potgieter & Engelbrecht, 2002, 2007).

The fitness function can either (1) evaluate the number of cases that are correctly classified for a decision tree, or (2) minimise the error between the predicted response of the individual and the target response of a number of training patterns. Additionally, the fitness function should implement a penalisation factor in order to penalise the complexity of a chromosome. In this manner, genetic programs can be used to minimise both the bias and the variance of the model described by a chromosome (Geman, Bienenstock, & Doursat, 1992).

*2.2.3. Scaling problems*

GP has shown considerable promise in its problem solving ability over a wide range of applications, including data mining (Geom, 1999; Kaboudan, 1999). However, problems exists in scaling GP to larger problems such as data mining. Marmelstein and Lamont summarise many of these difficulties (Marmelstein & Lamont, 1998). Some of the most important scaling problems are:

- GP performance is very dependent on the composition of the function and terminal sets.
- There is a trade-off between GP's ability to produce good solutions and parsimony.
- The size and complexity of GP solutions can make it difficult to understand. Furthermore, solutions can become bloated with extraneous code (also known as introns).

Of the above difficulties, the most difficult to control is the complexity of GP solutions, otherwise known as code growth. Abass describes many methods for the removal of introns, *e.g.* chromosome parsing, alternative selection methods and alternative crossover methods (Abass et al., 2002). Rouwhorst et al. presented a building-block approach to evolve decision trees (Engelbrecht, Schoeman, & Rouwhorst, 2001). In this approach, the initial population consists of very simple individuals which grows in complexity when their current structure is not sufficient to reduce the model error. Individuals are grown by adding a random generated building-block to their current structure.

## 3. GPMCC structure

This section discusses a genetic programming approach for mining continuous-valued classes (GPMCC). Section 3.1 presents an overview of the GPMCC method and its various components. An iterative learning strategy used by the GPMCC method is discussed in Section 3.2. Section 3.3 describes, in detail, the fragment pool utilised by the GPMCC method. Finally, the core genetic program for model tree induction is discussed in Section 3.4.

### 3.1. Overview

The genetic program for the mining of continuous-valued classes (GPMCC) consists of three parts:

1. An iterative learning strategy to reduce the number of training patterns that are presented to the genetic program.
2. A pool of non-linear fragments evolved using a GA (Potgieter & Engelbrecht, 2002, 2007), which serve as a terminal set for the terminal nodes of a chromosome in a genetic program. This pool of fragments is evolved using mutation and crossover operators.
3. A genetic program to evolve model trees.

Fig. 2 shows an overview of the GPMCC learning process. In addition, the GPMCC learning process is summarised below:

1. Let $g = 0$ be the generation counter
2. Initialise a population $G_{GP,g}$ of $N$ individuals, *i.e.*
   $G_g = \{I_\chi | \chi = 1, \ldots, N\}$
3. While $g < M$, where $M$ is the total number of generations, do
   (a) Sample $S$, using an incremental training strategy, from the remaining training patterns $P$, *i.e.* $P' \subset P$, $S = S \cup P'$.
   (b) Remove the sampled patterns from $P$, *i.e.* $P = P/S$.
   (c) Evaluate the fitness $R^2_a(I_\chi)$ of each individual in population $G_{GP,g}$ using the patterns in $S$.
   (d) Let $G'_{GP,g} \subset G_{GP,g}$ be the top $x\%$ of the individuals, based on fitness, to be involved in elitism.
   (e) Install the members of $G'_{GP,g}$ into the new generation $G_{GP,g+1}$.
   (f) Let $G''_{GP,g} \subset G_{GP,g}$ be the top $n\%$ of the individuals, based on fitness, to be involved in crossover.
   (g) Run the fragment pool optimisation algorithm once.
   (h) Perform crossover:
      (i) Randomly select two individuals $I_\alpha$ and $I_\beta$ from $G''_{GP,g}$.
      (ii) Produce offspring $I_\gamma$ from $I_\alpha$ and $I_\beta$.
      (iii) Install $I_\gamma$ into $G'''_{GP,g}$.
   (i) Perform mutation:
      (i) Select an individual $I_\omega$ from $G'''_{GP,g}$.
      (ii) Mutate $I_\omega$ by randomly selecting a mutation operator to perform.
      (iii) Install $I_\omega$ into the new generation $G_{GP,g+1}$.
   (j) Evolve the next generation $g := g + 1$.

### 3.2. Iterative learning strategy

The GPMCC method utilises an iterative learning strategy to reduce the number of training pattern presentations per generation. Additionally, the iterative learning strategy should result in more accurate rules being generated (Engelbrecht & Brits, 2002; Quinlan, 1993). The strategy utilises the $k$-means clustering algorithm (Forgy, 1965) to cluster the training data.

A *stratified random sample* is selected from the available training patterns during each generation of the genetic program. The size, $s$, of the initial sample is chosen as a percentage of the total number of training patterns $|P|$. The sampling strategy utilises an acceleration rate to increase the size of the sample during every generation of the genetic program. This size of the sample is increased until it equals the total number of training patterns.

The stratified random sample is drawn proportionally from each of the $k$ clusters of the $k$-means clusterer, *i.e.*:

$$S = S \cup_{\delta=1}^{k} C'_\delta$$

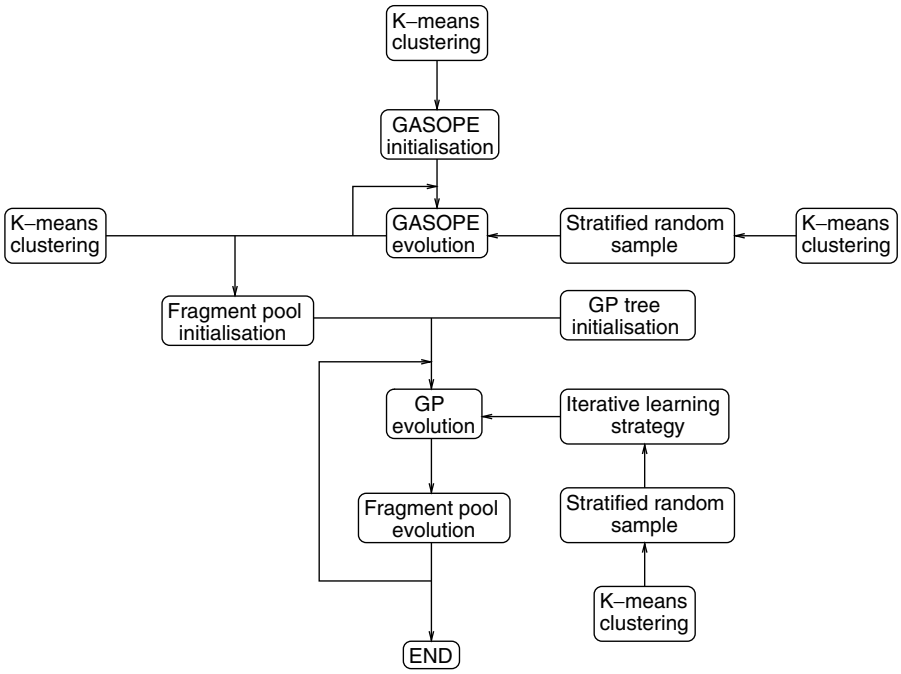


Fig. 2. Overview of the GPMCC learning process.

where

$$C'_\delta \subset C_\delta : |C'_\delta| = \frac{|C_\delta| \cdot acceleration \cdot s}{|P|}$$

and $|C_\delta|$ is the (stratum) size of cluster $C_\delta$, $|P|$ is the size of the data set, $s$ is the initial sample size and *acceleration* is the acceleration rate. The acceleration rate increases the size of the sample at each generation.

### 3.3. The fragment pool

A model tree is a decision tree that implements multi-variate linear models at its terminal nodes. However, linear models may not adequately describe time-series data or non-linear data. In these cases, a model tree that implements multi-variate linear models at its terminal nodes will perform a piecewise approximation of the problem space. Although such a piecewise approximation may be accurate, the number of rules induced by the approximation will be large as indicated by the results in Section 4.3.

A number of techniques exist to perform non-linear regression. Two obvious non-linear regression techniques from the evolutionary computation field include

- using a genetic program to perform a symbolic regression of the data covered by the terminal nodes,
- or using a GA, such as the GASOPE method in Potgieter and Engelbrecht (2002, 2007) to perform a non-linear regression of the data patterns covered by each terminal nodes.

The GPMCC discussed in this section uses GASOPE (Potgieter & Engelbrecht, 2002, 2007) as the approach to perform non-linear regression. However, the use of both of these methods can be shown to have a severe impact on the time taken to construct a model tree.

As reported in Potgieter (2003) and Potgieter and Engelbrecht (2007), the largest training time of the GASOPE method was approximately 1.5 s (considering a variety of problems). Assuming a genetic program is used to construct a model tree, if a model is generated by a mutation operator, a 1.5 s time penalty will be incurred every time that mutation operator is called. If, for example, there is a 1% chance of the mutation operator being run on an individual in a population of 100 individuals with, on average, 5 terminal nodes per individual, a 7.5 s time penalty will be incurred per generation ($0.01 \times 100 \times 5 \times 1.5 = 7.5$). For 1000 generations this performance penalty is 7500 s (2 h and 5 min). In other words, a very large proportion of the genetic program's training time will be spent optimising the models.

This section discusses the *fragment pool*, which is a collection of multi-variate non-linear models. The fragment pool is an evolutionary algorithm for improving the time taken for a model to be generated, based on context modelling. The fragment pool represents a *belief space* of terminal symbols for a model tree. The remainder of this section discusses the *representation* of the fragment pool, the *initialisation* of a fragment, the fragment *mutation* and *crossover* operators, and the *fitness function*.

The implementations of the fragment pool and the genetic program of Section 3.4 to evolve model trees are

heavily intertwined. Therefore, for the remainder of this section assume that there is a genetic program that evolves model trees, whose terminal nodes are models that are obtained from the fragment pool.

*3.3.1. Representation*

Each *fragment* in the pool represents a model-lifetime mapping

$$F_\omega = \{I_\omega \rightarrow \pi_\omega\}$$

where $I_\omega$ is a GASOPE individual (i.e. a multi-variate non-linear model) and $\pi_\omega$ is the lifetime of $I_\omega$. The lifetime $\pi_\omega$ represents the age of a fragment in the fragment pool. When a fragment's lifetime expires, it is removed from the pool. The lifetime of a fragment can, however, be reset if the fragment is deemed "useful". By counting the number of times a model appears as a terminal node in the members of the crossover group of the genetic program, the fragment usefulness can be determined. A model is more likely to appear as a terminal node of members of the crossover group if the model closely approximates the subspace described by the training and validation patterns covered by the path to the terminal node. Thus, the fragment pool implements a kind of *context modelling* (Salomon, 2000), because fragments that result in sub-optimal approximations for these subspaces are eventually removed and can no longer contaminate the pool.

*3.3.2. Initialisation*

The initialisation of the fragment pool $G_{FP}$ proceeds as follows:

1. Let $k = initial\_clusters$.
2. Perform $k$-means clustering to obtain $k$ clusters $C_\delta$, $\delta = 1,\ldots,k$.
3. For each cluster $C_\delta$, $\delta = 1,\ldots,k$.
   (a) Use the GASOPE method to obtain a non-linear regression $I_\delta$ of the patterns in each cluster.
   (b) Insert the fragment $F_\delta = \{I_\delta \rightarrow 0\}$ into the fragment pool $G_{FP} = G_{FP} \cup F_\delta$.
4. Divide $k$ by the split factor *i.e* $k := \frac{k}{split\_factor}$.
5. Return to step 2 while $k > 1$.

Essentially, the above algorithm performs multiple piecewise approximations of the problem space to build an initial set of fragments. The number of initial clusters, *initial_clusters*, and the split factor, *split_factor*, ultimately control the total number of piecewise approximations (models) that are generated. The algorithm starts by fitting many highly specific approximations and then increasing the approximation generality by decreasing the available number of clusters $k$. Decreasing the available number of clusters results in an increase in the number of patterns covered by each cluster centroid. This, in turn, results in a more general function approximation per cluster.

The models contained within the fragments form part of a terminal set for the genetic program. Whenever the genetic program requires a model, the fragment pool randomly selects a fragment and passes the fragment's model to the genetic program.

*3.3.3. Mutation and crossover operators*

The fragment pool mutation operators serve to inject additional fragments into the fragment pool. The addition of fragments to the fragment pool result in an increased number of models in the terminal set of the genetic program. Additional models in the terminal set prevent the stagnation of the genetic program, by allowing the introduction of models that approximate regions not covered by the initialisation of the fragment pool. Additionally, the mutation operators also serve to fine-tune the models at the terminal nodes of the genetic program. Two mutation operators exist for the fragment pool: The shrink operator (Potgieter & Engelbrecht, 2002, 2007) duplicates an arbitrary fragment in the fragment pool to remove, arbitrarily, one of the term-coefficient pairs from the fragment. The introduce operator calls the GASOPE optimisation algorithm (Potgieter & Engelbrecht, 2002, 2007) on the training and validation patterns covered by the *path* of a terminal node of an arbitrary individual of the genetic program to evolve a non-linear model. The path is defined as the nodes traversed from the root of a tree to a specific node in the tree. The model obtained from the GASOPE optimisation algorithm is given a lifetime of 0 and is inserted into the fragment pool.

The crossover operator of the fragment pool is an invocation of the GASOPE crossover operator (Potgieter & Engelbrecht, 2002, 2007). A fragment with a non-zero usefulness factor and an arbitrary fragment are randomly chosen from the fragment pool and their models are given to the GASOPE crossover operator. The model obtained from the GASOPE crossover operator is given a lifetime of 0 and is inserted into the fragment pool. The reader is referred to (Potgieter & Engelbrecht, 2007) for more detail.

A culling operator removes fragments from the fragment pool, when those fragment's fragment lifetimes have expired, *i.e.* when the fragment lifetimes are larger than some upper-bound. The culling operator removes fragments from the fragment pool that have not been useful for a number of generations. This operator ensures that the fragment pool does not become uncontrollably large.

The shrink operator and the crossover operator are uniformly, randomly applied once after the completion of a generation of the genetic program. This ensures that the size of the fragment pool increases at least once per generation, in order to counteract the effects of the fragment lifetime, *i.e.* the removal of useless fragments. The introduce operator is applied with a statistical probability whenever

the genetic program requires a model, *i.e.* when the relevant genetic program mutation operator is invoked. If the fragment lifetime is too large or the introduce operator is applied too often, the fragment pool will grow too quickly.

A large fragment pool results in a large number of terminal symbols, which may have a negative consequence on the convergence properties of the genetic program. Conversely, a small fragment pool may lead to the stagnation of the genetic program, because there may not be enough variation in the terminal symbols described by the fragment pool.

#### 3.3.4. Fitness function

The fitness function rewards the usefulness of a fragment. As was mentioned earlier, the fragment usefulness is determined by counting the number of times a fragment appears as a terminal node of individuals in the crossover group of the genetic program. The crossover group consists of the top individuals in the genetic program, obtained through tournament selection. Thus, the usefulness of a fragment is determined by the number of times it appears as well as where it appears, *i.e.* fragments not used as terminal symbols of the crossover group are useless, because the overall fitness of the individuals using those fragments is poor.

#### 3.3.5. Fragment pool optimisation algorithm

The optimisation algorithm for the fragment pool is as follows:

1. Obtain a set $G''_{GP,g}$ of individuals from the crossover group of a genetic program.
2. Evaluate the fitness of each fragment $F_\omega$ in the pool $G_{FP}$ using $G''_{GP,g}$, *i.e.* for each fragment $F_\omega \in G_{FP}$:
   (a) Count the number of times, $n$, that the individual $I_\omega$ appears as a terminal symbol in $G''_{GP,g}$.
   (b) Set the fitness of the individual $F_{FP}(F_\omega) = n$.
3. Select a crossover group from the pool $G'_{FP} \subset G_{FP}$, where $F_{FP}(F_\omega) > 0, F_\omega \in G'_{FP}$ (the fragments of $G'_{FP}$ still reside in $G_{FP}$).
4. Reset the fragment lifetime of all fragments in $GP'_{FP}$ to 0, as they were deemed useful.
5. Increase the fragment lifetime of all fragments in $GP_{FP}/GP'_{FP}$ by one.
6. Remove/cull any fragment from $G_{FP}$ whose fragment lifetime has expired.
7. If $U(0,1) < 0.5$
   (a) Select a fragment $F_\alpha$ from $G'_{FP}$ and a fragment $F_\beta$ from $G_{FP}$.
   (b) Perform crossover to obtain a fragment $F_\gamma = \{I_\gamma \rightarrow 0\}$.
   (c) Insert $F_\gamma$ into $G_{FP}$
8. Otherwise,
   (a) Select a fragment $F_\omega$ from $G_{FP}$.
   (b) Duplicate $F_\omega$ to get $F'_\omega$.
   (c) Perform mutation on $F'_\omega$.
   (d) Insert $F'_\omega$ into $G_{FP}$.

The fragment pool optimisation algorithm is invoked at each generation by the genetic program.

### 3.4. Genetic program for model tree induction

This section discusses, in detail, a genetic program for inducing model trees. The models for this genetic program are obtained from the fragment pool discussed in Section 3.3.

#### 3.4.1. Representation

Each individual $I_\chi$ in the population represents a model tree such that:

$I_\chi = NODE$

where

$$
\begin{aligned}
&NODE:\\
&(CONSEQUENT)|((ANTECEDENT \rightarrow NODE)\\
&\quad \vee (\neg ANTECEDENT \rightarrow NODE))\\
&ANTECEDENT: (NOMINAL\_ANTECEDENT)\\
&\quad |(CONTINUOUS\_ANTECEDENT)\\
&NOMINAL\_ANTECEDENT: (A_\xi = v_\xi)\\
&CONTINUOUS\_ANTECEDENT:\\
&(A_\xi < v_\xi)|(A_\xi > v_\xi)|(A_\xi = v_\xi)|(A_\xi \neq v_\xi)\\
&CONSEQUENT: (I_\omega)
\end{aligned}
$$

A consequent, $I_\omega$, represents a GASOPE model from the fragment pool, $A_\xi$ represents a nominal-, continuous- or discrete-valued attribute and $v_\xi$ represents a possible value of $A_\xi$. For the continuous antecedents, operators such as $\leqslant$ and $\geqslant$ are obtained by adjusting the attribute value $v_\xi$.

#### 3.4.2. Initialisation

The GPMCC initialise operator creates an individual $I_\chi$ by recursively adding nodes to that individual, up to a maximum depth bound. The pseudo-code algorithm for the initialisation is as follows, where *CALLER* is a calling node initially set to *Nil* and *depth* is the maximum required depth of the tree:

**Initialise**: with parameters *CALLER* and *depth*

1. If $depth < maximum\_depth$ and $U(0,1) < 0.5$
   (a) Select an attribute $A_\xi$, $\xi = 1, \ldots, I$ from the attribute space of dimension $I$.
   (b) If $A_\xi$ is a continuous-valued attribute, select an operator $\text{op}(\xi) \in \{<, >, =, \neq\}$.
   (c) Otherwise, $\text{op}(\xi) \in \{=\}$.
   (d) Select an attribute value $a_{\xi,i}$ for attribute $A_\xi$ from a training pattern $i$, such that $v_\xi = a_{\xi,i}$, $i \in \{1, \ldots, |P|\}$.
   (e) Create a node $N_\chi$ with antecedent $ant_\chi = (A_\xi \text{op}(\xi) v_\xi)$ and consequent $con_\chi = Nil$.
   (f) Call **Initialise** with the node covered by the antecedent of $N_\chi$ (the left node), $N_{ant_\chi}$, and depth $depth + 1$.
   (g) Call **Initialise** with the node covered by the negation of the antecedent of $N_\chi$ (the right node), $N_{\neg ant_\chi}$, and depth $depth + 1$.

2. Otherwise,
   (a) Select an individual $I_\omega$ from the fragment pool.
   (b) Create a node $N_\chi$ with antecedent $ant_\chi = Nil$ and consequent $con_\chi = I_\omega$.
   (c) Set the node covered by the antecedent of $N_\chi$:

   $$N_{ant_\chi} := Nil$$

   (d) Set the node covered by the negation of the antecedent of $N_\chi$:

   $$N_{\neg ant_\chi} := Nil$$

3. Let $CALLER := N_\chi$ and return.

Once the procedure terminates, *CALLER* returns with the head of the tree. Fig. 3 illustrates one outcome of the initialisation of an individual $I_\chi$. Each node in the diagram is recursively initialised as $N_\chi$ and the arrows show the path taken by the initialisation method.

### 3.4.3. Mutation operators

The mutation operators serve to inject new genetic material into the population. Additionally, the mutation operators utilise domain specific knowledge in order to improve the quality of the individuals in the population. A large number of mutation operators have been developed for the GPMCC method.

- **Expand-worst-terminal-node operator:** The expand-worst-terminal-node operator locates and partitions the subspace for which a terminal node has a higher relative error than all other terminal nodes in the individual $I_\chi$. The relative error of the terminal node is determined by using the mean squared error $E_{MS}$ between the model described by the terminal node and the training set covered by the path of that terminal node. The operator attempts to maximise the adjusted coefficient of determination $R^2_a$ (fitness) of the individual, by partitioning the subspace described by a terminal node into smaller subspaces. The pseudo-code for the expand-worst-terminal-node operator is as follows:

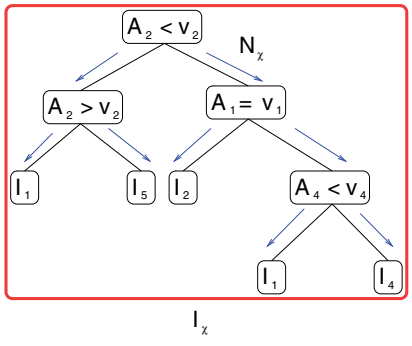


Fig. 3. Illustration of GPMCC chromosome for an individual $I_\chi$.

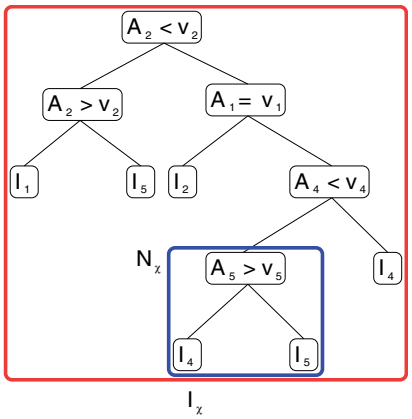


Fig. 4. Illustration of the expand-worst-terminal-node operator for an individual $I_\chi$.

1. Select $N_\chi$ (shown in Fig. 4) such that $\forall N_i \in I_\chi : (con_\chi \neq Nil) \wedge (E_{MS}(N_\chi) \leqslant \mathrm{E}_{MS}(N_i))$, *i.e.* select the worst terminal node.
2. Select an attribute $A_\xi$, $\xi = 1, \ldots, I$ from the attribute space of size $I$ (in order to turn the consequent into an antecedent).
3. If $A_\xi$ is a continuous-valued attribute, select an operator $\mathrm{op}(\xi) \in \{<, >, =, \neq\}$.
4. Otherwise, $\mathrm{op}(\xi) \in \{=\}$.
5. Select an attribute value $a_{\xi,i}$ for attribute $A_\xi$ from a training pattern $i$, such that $v_\xi = a_{\xi,i}$, $i \in \{1, \ldots, |P|\}$.
6. Set the antecedent $ant_\chi$ of node $N_\chi$ to $(A_\xi \mathrm{op}(\xi) v_\xi)$.
7. Set the consequent $con_\chi$ of node $N_\chi$ to $Nil$ (to satisfy the termination criteria).
8. Create a node covered by the antecedent of $N_\chi$ (the left node), $N_{ant_\chi}$, with antecedent $ant_{\neg ant_\chi} = Nil$ and consequent $con_{ant_\chi} = I_\omega$.
9. Set the node covered by the antecedent of $N_{ant_\chi}$:

   $$N_{ant_{ant_\chi}} := Nil$$

10. Set the node covered by the negation of the antecedent of $N_{ant_\chi}$:

    $$N_{ant_{\neg ant_\chi}} := Nil$$

11. Create a node covered by the negation of the antecedent of $N_\chi$ (the right node), $N_{\neg ant_\chi}$, with antecedent $ant_{\neg ant_\chi} = Nil$ and consequent $con_{\neg ant_\chi} = I_\omega$.
12. Set the node covered by the antecedent of $N_{\neg ant_\chi}$:

    $$N_{\neg ant_{ant_\chi}} := Nil$$

13. Set the node covered by the negation of the antecedent of $N_{\neg ant_\chi}$:

    $$N_{\neg ant_{\neg ant_\chi}} := Nil$$

Intuitively, the expand-worst-terminal-node operator attempts to increase the fitness of an individual, by partitioning the subspace covered by the worst terminal node (in terms of mean squared error) into two more subspaces. A high mean squared error is an indication of a poor function approximation. It is

possible that by partitioning the subspace the accuracy of the function approximation can be increased. Thus, this operator caters for discontinuities in the input space.

- **Expand-any-terminal-node operator:** The expand-any-terminal-node operator partitions the subspace of a random terminal node in an individual $I_\chi$. The pseudo-code for the expand-any-terminal-node operator is identical to the expand-worst-terminal-node operator, except that step (1) should read:
  1. Select $N_\chi$ from $I_\chi$ such that $con_\chi \neq Nil$.

  If the high mean squared error in the worst terminal node is due to a large variance in the data, the expand-worst-terminal-node operator will continually attempt to partition the subspace of the worse terminal node to no avail. This could lead to extremely slow convergence of the GPMCC method. The expand-any-terminal-node prevents this scenario from occurring by allowing any terminal node to be expanded.
- **Shrink operator:** The shrink operator replaces a non-terminal node of an individual $I_\chi$ with one of the non-terminal node's children. The pseudo-code for the shrink operator is as follows:
  1. Select $N_\chi$ (as shown in Fig. 5) from $I_\chi$ such that $(ant_\chi \neq Nil)$.
  2. If $U(0,1) < 0.5$ then the current node becomes the node covered by the antecedent (the left node) $N_\chi := N_{ant_\chi}$.
  3. Otherwise, the current node becomes the node covered by the negation of the antecedent (the right node) $N_\chi := N_{\neg ant_\chi}$ (as shown in Fig. 5).

  The shrink operator is responsible for removing introns from an individual, which is necessary to prevent code bloat.
- **Perturb-worst-non-terminal-node operator:** The perturb-worst-non-terminal-node operator selects and perturbs a non-terminal node which has a higher relative error than all other non-terminal nodes in an individual $I_\chi$. Once again, the relative error is determined using the mean squared error, $E_{MS}$, on the training set. This operator gives the GPMCC method an opportunity to optimise the partitions described by the non-terminal nodes of an individual. The pseudo-code for the perturb-worst-non-terminal-node operator is as follows:

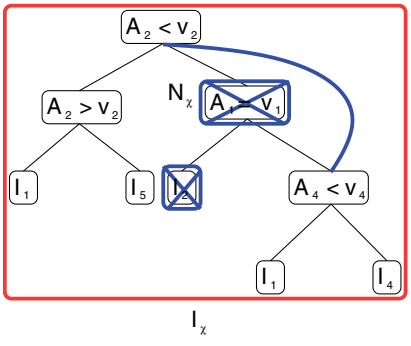


Fig. 5. Illustration of the shrink operator for an individual $I_\chi$.

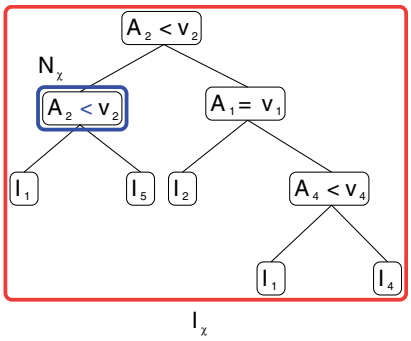


Fig. 6. Illustration of the perturb-worst-non-terminal-node operator for an individual $I_\chi$.

1. Select $N_\chi$ (as shown in Fig. 6) such that $\forall N_i \in I_\chi : (ant_\chi \neq Nil) \wedge (E_{MS}(N_\chi) \leqslant E_{MS}(N_i))$.
2. If $U(0,1) < U_1$, where $U_1 \in [0,1]$ is a user-defined parameter
   (a) If $A_\xi$ is a continuous-valued attribute
      (i) If $U(0,1) < U_2$ where $U_2 \in [0,1]$ is a user-defined parameter, select an operator $\text{op}(\xi) \in \{<, >, =, \neq\}$ (as shown in Fig. 6).
      (ii) Otherwise ($U(0,1) \geqslant U_2$), adjust the attribute value $v_\xi$ according to a Gaussian distribution $v_\xi := v_\xi + \frac{-(max-min)U(0,1)^2}{2U_3(0.3)^2}$, where $\forall a_{\xi,i}, i = 1,\ldots,I : (max \geqslant a_{\xi,i}) \wedge (min \leqslant a_{\xi,i}), U_3 \in \Re$ is a user-defined parameter, $min$ is the minimum value for an attribute $A_\xi$ and $max$ is the maximum value for an attribute $A_\xi$. The standard deviation 0.3 of the Gaussian distribution provides an even distribution of the Gaussian function in the domain $[0,1]$.
   (b) Otherwise ($A_\xi$ is not a continuous-valued attribute),
      (i) Randomly, select an attribute value $a_{\xi,i}$ for attribute $A_\xi$ from a training pattern $i$, and let $v_\xi = a_{\xi,i}$, $i \in \{1,\ldots,|P|\}$.
3. Otherwise ($U(0,1) \geqslant U_1$),
   (a) Select an attribute $A_\xi$, $\xi = 1,\ldots,I$ from the attribute space of size $I$.
   (b) If $A_\xi$ is a continuous-valued attribute, select an operator $\text{op}(\xi) \in \{<, >, =, \neq\}$.
   (c) Otherwise, $\text{op}(\xi) \in \{=\}$.
   (d) Randomly select an attribute value $a_{\xi,i}$ for attribute $A_\xi$ from a training pattern $i$, such that $v_\xi = a_{\xi,i}$, $i \in \{1,\ldots,|P|\}$.
4. Set the antecedent $ant_\chi$ of node $N_\chi$ to $(A_\xi \ \text{op}(\xi) \ v_\xi)$.

Intuitively, the partition described by a non-terminal node of an individual may not correctly partition the subspace *e.g.* if a test should have been $A_1 < 5$, but is actually $A_1 < 4.6$. The perturb-worst-non-terminal-node operator specifically attempts to adjust the test described by the worst non-terminal node (indicated by the largest mean squared error).

- **Perturb-any-non-terminal-node operator:** The perturb-any-non-terminal-node operator selects and perturbs a non-terminal node in an individual $I_\chi$. The perturb-any-non-terminal-node operator is identical to the perturb-worst-non-terminal-node operator except that step (1) should read:
  1. Select $N_\chi$ such that $(ant_\chi \neq Nil)$.

  This operator allows for the perturbation of any non-terminal node, in order to prevent slow convergence in the case that the high mean squared error of the worst non-terminal node is due to high variation in the dataset.
- **Perturb-worst-terminal-node operator:** The perturb-worst-terminal-node operator selects and perturbs a terminal node which has a higher relative error than all other non-terminal nodes in an individual $I_\chi$. This operator gives the GPMCC method an opportunity to optimise the non-linear approximations for the subspace covered by the training and validation patterns described by the path to a terminal node. The perturb-worst-terminal-node operator is as follows:
  1. Select $N_\chi$ (as shown in Fig. 7) such that $\forall N_i \in I_\chi : (con_\chi \neq Nil) \wedge (E_{MS}(N_\chi) \leqslant \mathrm{E}_{MS}(N_i))$.
  2. Select an individual $I_\omega$ from the fragment pool.
  3. Set the consequent $con_\chi$ of node $N_\chi$ to $I_\omega$.

  Intuitively, the model described by the terminal node of an individual may be a poor fit of the data covered by the path of that terminal node. The perturb-worst-terminal-node operator randomly selects a new individual from the fragment pool to replace the current model.
- **Perturb-any-terminal-node operator:** The perturb-any-terminal-node operator selects and perturbs a terminal node in an individual $I_\chi$. The perturb-any-terminal-node operator is identical to the perturb-worst-terminal-node operator except that step (1) should read:
  1. Select $N_\chi$ such that $(con_\chi \neq Nil)$.

  This operator allows for the perturbation of any terminal node, in order to prevent slow convergence in the case that the high mean squared error of the worst terminal node is due to high variation in the dataset.
- **Reinitialise operator**: The reinitialise operator is a re-invocation of the initialisation operator.

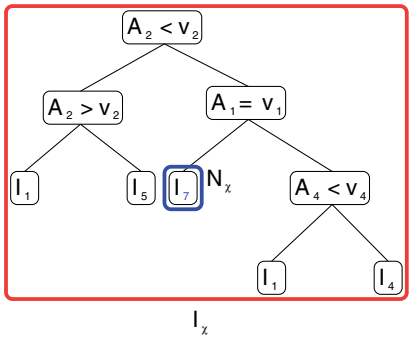


Fig. 7. Illustration of the perturb-worst-terminal-node operator for an individual $I_\chi$.

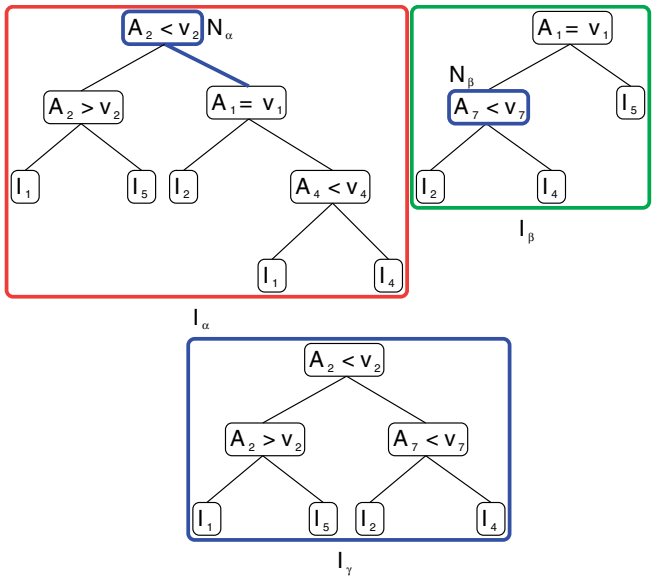


Fig. 8. Illustration of the crossover operator for individuals $I_\alpha$, $I_\beta$ and $I_\gamma$.

### 3.4.4. Crossover operator

The crossover operator implements a standard genetic program crossover strategy. Two individuals ($I_\alpha$ and $I_\beta$) are chosen by tournament selection from the population and a crossover point is chosen for each individual. The two crossover points are spliced together to create a new individual $I_\gamma$. The pseudo-code for the crossover operator is as follows (as illustrated in Fig. 8):

1. Select an individual $I_\alpha \in G_{GP}$ from the population $G_{GP}$.
2. Select an individual $I_\beta \in G_{GP}$ from the population $G_{GP}$.
3. Select a non-terminal node $N_\alpha$ from $I_\alpha$.
4. Select a node $N_\beta$ from $I_\beta$.
5. Create a new individual $I_\gamma$ from $I_\alpha$ and $I_\beta$ by installing $N_\beta$ as a child of $N_\alpha$.

### 3.4.5. Fitness function

As for all evolutionary computation paradigms, the fitness function is a very important aspect of a genetic program, in that it serves to direct the algorithm toward optimal solutions. The fitness function used by the genetic program is an extended form of the *adjusted coefficient of determination*:

$$R_a^2 = 1 - \frac{\sum_{i=1}^{s}(b_i - b^{\bullet}_{I_\chi,i})^2}{\sum_{i=1}^{s}(b_i - \overline{b}_i)^2} \cdot \frac{s-1}{s-k} \tag{4}$$

where $s$ is the size of the sample set, $b_i$ is the actual output of pattern $i$, $b^{\bullet}_{I_\chi,i}$ is the predicted output of individual $I_\chi$ for pattern $i$, and the model complexity $d$ is calculated as follows:

$$d = \sum_{\mu=1}^{|I_\chi|} \begin{cases} 1 + \sum_{\xi=1}^{|I_\omega|} \sum_{\tau=1}^{|T_\xi|} \lambda_{\xi,\tau} & \text{if } con_\mu \neq Nil \\ 1 & \text{if } con_\mu = Nil \end{cases} \quad (5)$$

where $I_\chi$ is an individual in the set $G_{GP}$ of individuals, $|I_\chi|$ is the number of nodes in $I_\chi$, $I_\omega$ is the model at a terminal node, $T_\xi$ is a term of $I_\omega$ and $\lambda_{\xi,\tau}$ is the order of term $T_\xi$. This fitness function penalises the complexity of an individual $I_\chi$ by penalising the size of an individual and the complexity of each of the non-terminal nodes of that individual, *i.e.* the number of nodes and the complexity of each leaf node, given as (Potgieter, 2003)

$$d = \sum_{\xi=1}^{|I_\chi|} \sum_{\tau=1}^{|T_\xi|} \lambda_{\xi,\tau} \quad (6)$$

## 4. Experimental results

This section discusses the experimental procedure and results of the GPMCC method, compared with NeuroLinear and Cubist, as applied to various data sets obtained from the UCI machine learning repository and a number of artificially created datasets. Section 4.1 presents the various data sets. The experimental procedure and parameter initialisation are discussed in Section 4.2. Section 4.3 presents the experimental results for the functions listed in Section 4.1. The quality of the generated rules is discussed in Section 4.4.

### 4.1. Datasets

The GPMCC method was evaluated on a number of benchmark approximation databases from the machine learning repository as well as a number of artificial databases (Blake, Keogh, & Merz, 1998). Table 1 describes each of the UCI databases used in this paper. The artificial databases were created to analyse various approximation problems not sufficiently covered by the UCI machine learning repository, *e.g.* time-series, and to provide a number of large databases, exceeding 10,000 patterns, with which to analyse the performance of the GPMCC method.

The function describing the *Machine* data set in Table 1 is piecewise linear. Therefore, the GPMCC method is not expected to perform substantially better, in terms of the number of rules generated, than other methods.

In addition to the problems listed in Table 1, the following problems were also used:

- The **function example** database represents a discontinuous function, defined by a number of standard polynomial expressions:

$$y = U(-1,1) + \begin{cases} 1.5x^2 - 7x + 2 & \text{if } (type = \text{`}C\text{'}) \wedge (x > 1) \\ x^3 + 400 & \text{if } (type = \text{`}C\text{'}) \wedge (x \leqslant 1) \\ 2x & \text{if } (type = \text{`}A\text{'}) \\ -2x & \text{if } (type = \text{`}B\text{'}) \end{cases}$$

  where $x \in [-10, 10]$ and $type \in \{\text{`}A\text{'}, \text{`}B\text{'}, \text{`}C\text{'}\}$. The database consists of 1000 patterns, with three attributes per pattern.
- The **Lena Image** database represents a $128 \times 128$ greyscale version of the famous "Lena" image, which is used for comparing different image compression techniques (Salomon, 2000). The data consists of 11 continuous-valued attributes, which represents a context (or footprint) for a given pixel. The objective of an approximation method is to infer rules between the context pixels and the target pixel. The database is large and consists of 16,384 patterns.
- The **Mono Sample** database represents an approximately 4 s long sound clip, sampled in mono at 8000 hertz (the chorus of U2's "Pride (In the name of love)"). The database consists of 31,884 patterns, with five attributes per pattern. The objective of an approximation method is to infer rules between a sample and a context of previous samples.
- The **Stereo Sample** database represents an approximately 4 s long sound clip, sampled in stereo at 8000 hertz (the chorus of U2's "Pride (In the name of love)"). The database consists of 31,430 patterns, with nine attributes per pattern. The objective of an approximation method is to infer rules between a sample in the left channel and a number of sample points in the left and right channels.
- The **Time-series** database represents a discontinuous application of components of the Rossler and Lorenz attractors, the Henon map and a polynomial term:

Table 1
Databases obtained from the UCI machine learning repository (Attributes: N = Nominal, C = Continuous)

| Dataset | Samples | Attributes | Prediction task |
|---|---|---|---|
| Abalone | 4177 | 1N, 7C | Age of abalone specimens |
| Auto-mpg | 392 | 7C | Car fuel consumption in miles per gallon |
| Elevators | 16,599 | 18C | Action taken for controlling an F16 aircraft |
| Federal Reserve Economic Data | 1049 | 16C | 1-Month credit deficit rate |
| Housing | 506 | 13C | Median value of homes in Boston suburbs |
| Machine | 209 | 6C | Relative CPU performance |
| Servo | 167 | 2N, 2C | Response time of a servo mechanism |

$$w_{n+1} = x_n y_n z_n$$
$$x_{n+1} = 1 - a \cdot x_n^2 + b \cdot y_n$$
$$y_{n+1} = x_n - a \cdot y_n$$
$$z_{n+1} = x_n y_n - b \cdot z_n$$
$$p = U(0,1) + \begin{cases} x_{n+1} & \text{if } (x_n \geqslant 0) \\ y_{n+1} & \text{if } (x_n < 0) \wedge (y_n \geqslant 0) \\ z_{n+1} & \text{if } (x_n < 0) \wedge (y_n < 0) \wedge (z_n \geqslant 0) \\ w_{n+1} & \text{if } (x_n < 0) \wedge (y_n < 0) \wedge (z_n < 0) \end{cases}$$

where $x_0, y_0, z_0, w_0 \sim U(-5,5)$. The database consists of 1000 patterns, with four attributes per pattern, generated using the Runga–Kutta method with order 4 (Burden & Faires, 1997).

### 4.2. Experimental procedure

Each of the databases was split up into a training set, a validation set and a generalisation set. The training set was used to train the GPMCC method, the validation set was used to validate the models of the GASOPE method and the generalisation set was used to test the performance of the GPMCC method on unseen data. The training set for each database consisted of roughly 80% of the patterns, with the remainder of the patterns split evenly among the validation and generalisation sets (80%:10%:10%). The GPMCC initialisation as used for each of the databases is shown in Table 2. For all datasets the maximum polynomial order was set to 5, except for the house-16H dataset, for which the maximum polynomial order was set to 10.

Generally speaking, the parameters prefixed by "Function" in Table 2 control the behaviour of the model optimisation algorithm of the fragment pool (Potgieter, 2003). These parameters are soft options for the GPMCC method, because they are automatically adjusted if they violate any of the restrictions of the GASOPE method. The parameters prefixed by "Decision" control the behaviour of the genetic program for generating model trees. In general, "consequent" refers to terminal nodes and "antecedent" refers to non-terminal nodes. The mnemonic "CA" stands for continuous antecedent, "NA" stands for nominal antecedent, "ME" stands for mutate expand and "MN" stands for mutate node (perturb).

The GPMCC method utilises a variable mutation rate. The mutation rate is initially set to a default parameter ("DecisionMutationRateInitial"). Every time the accuracy of the best individual in a generation does not increase, the mutation rate is increased (by the amount specified by "DecisionMutationRateIncrement") up to a maximum mutation rate (given by "DecisionMutationRateMax").

This variable mutation rate helps to prevent stagnation. If the accuracy of the best individual does not improve over a number of generations, the increase in mutation rate injects more new genetic material into the population. However, if the accuracy of the best individual does improve, then the mutation rate is reset to the initial mutation rate.

Table 2
GPMCC initialisation parameters

| Parameter | Value |
|---|---|
| SyntaxMode | 1 |
| Clusters | 30 |
| ClusterEpochs | 10 |
| FunctionMutationRate | 0.1 |
| FunctionCrossoverRate | 0.2 |
| FunctionGenerations | 100 |
| FunctionIndividuals | 30 |
| PolynomialOrder | 5 |
| FunctionPercentageSampleSize | 0.01 |
| FunctionMaximumComponents | 10 |
| FunctionElite | 0.1 |
| FunctionCutOff | 0.001 |
| DecisionMaxNodes | 30 |
| DecisionMEWorstVsAnyConsequent | 0.5 |
| DecisionMECreateVsRedistributeLeafNodes | 0.5 |
| DecisionMNAntecedentVsConsequent | 0.5 |
| DecisionMNWorstVsAnyAntecedent | 0.5 |
| DecisionMNWorstVsAnyConsequent | 0.5 |
| DecisionReoptimizeVsSelectLeaf | 0.1 |
| DecisionMutateExpand | 0.3 |
| DecisionMutateShrink | 0.3 |
| DecisionMutateNode | 0.3 |
| DecisionMutateReinitialize | 0.1 |
| DecisionNAAttributeVsClassOptimize | 0.2 |
| DecisionCAAttributeOptimize | 0.1 |
| DecisionCAClassOptimize | 0.6 |
| DecisionCAConditionOptimize | 0.3 |
| DecisionCAClassVsGaussian | 0.1 |
| DecisionCAClassPartition | 0.1 |
| DecisionCAConditionalPartition | 0.1 |
| DecisionPoolNoClustersStart | 30 |
| DecisionPoolNoClustersDivision | 2 |
| DecisionPoolNoClusterEpochs | 1000 |
| DecisionPoolFragmentLifeTime | 50 |
| DecisionInitialPercentageSampleSize | 0.1 |
| DecisionSampleAcceleration | 0.005 |
| DecisionNoIndividuals | 100 |
| DecisionNoGenerations | 10 |
| DecisionElite | 0.0 |
| DecisionMutationRateInitial | 0.2 |
| DecisionMutationRateIncrement | 0.01 |
| DecisionMutationRateMax | 0.6 |
| DecisionCrossoverRate | 0.1 |
| CrossValidation | 0 |

The influence of GPMCC parameters on performance was empirically investigated in Potgieter (2003), where it was shown that performance is insensitive to different parameter values.

### 4.3. Method comparison

Although a large number of initial parameters were introduced in the previous section, the GPMCC method appears to be fairly robust in that different values for parameters do not have a significant effect on accuracy (Potgieter, 2003). This section compares the GPMCC

Table 3
Comparison of Cubist, GPMCC and NeuroLinear

| Dataset | Method | $\overline{GMAE}$ | $\sigma_{GMAE}$ | $\overline{rules}$ | $\sigma_{rules}$ | $\overline{conds}$ | $\sigma_{conds}$ | $\overline{terms}$ | $\sigma_{terms}$ |
|---|---|---|---|---|---|---|---|---|---|
| Abalone | Cubist | 1.4950 | 0.0609 | 12.6500 | 4.1056 | 2.6545 | 0.5700 | 4.7560 | 0.4978 |
| | GPMCC | 1.6051 | 0.0156 | 2.5400 | 0.6264 | 1.3407 | 0.3688 | 8.3213 | 1.0634 |
| | NL | 1.5700 | 0.0600 | 4.1000 | 1.4500 | n/a | n/a | n/a | n/a |
| Auto-Mpg | Cubist | 1.8676 | 0.2536 | 5.1000 | 1.3890 | 2.0816 | 0.4225 | 3.0013 | 0.4972 |
| | GPMCC | 2.1977 | 0.3703 | 2.1800 | 0.6724 | 1.0765 | 0.4985 | 7.2195 | 1.6180 |
| | NL | 1.9600 | 0.3200 | 7.5000 | 5.0800 | n/a | n/a | n/a | n/a |
| Housing | Cubist | 1.7471 | 0.2633 | 12.6700 | 3.3727 | 3.2125 | 0.4542 | 5.2164 | 0.6068 |
| | GPMCC | 2.8458 | 0.5217 | 2.8200 | 0.7962 | 1.5103 | 0.4534 | 7.6613 | 1.4220 |
| | NL | 2.5300 | 0.4600 | 25.3000 | 17.1300 | n/a | n/a | n/a | n/a |
| Machine | Cubist | 26.9280 | 7.5873 | 4.8800 | 1.4162 | 1.9367 | 0.4106 | 4.6818 | 0.7785 |
| | GPMCC | 34.3228 | 17.0849 | 3.4600 | 0.9773 | 1.8770 | 0.4983 | 3.4095 | 0.9773 |
| | NL | 20.9900 | 11.3800 | 3.0000 | 3.0000 | n/a | n/a | n/a | n/a |
| Servo | Cubist | 0.3077 | 0.1252 | 9.6100 | 1.9638 | 2.7298 | 0.3027 | 2.1033 | 0.2890 |
| | GPMCC | 0.4496 | 0.1755 | 5.1500 | 1.9456 | 2.6247 | 0.8876 | 2.7935 | 0.8482 |
| | NL | 0.3400 | 0.0800 | 4.7000 | 2.3100 | n/a | n/a | n/a | n/a |

$\overline{GMAE}$ is the average generalisation mean absolute error, $\sigma_{GMAE}$ is the standard deviation for the generalisation mean absolute error, $\overline{rules}$ represents the average number of rules, $\sigma_{rules}$ represents the standard deviation for the number of rules, $\overline{conds}$ represents the average number of rule conditions, $\sigma_{conds}$ is the standard deviation for the number of rule conditions, $\overline{terms}$ represents the average number of rules per term and $\sigma_{terms}$ is the standard deviation for the number of rules per term.

method to two other methods discussed earlier in this paper.

The first comparison method is Setiono's NeuroLinear method from Section 2.1. Setiono presents a table of results, obtained by running NeuroLinear on five databases from the UCI machine learning repository (Setiono, 2001). The databases used by Setiono are the Abalone, Auto-Mpg, Housing, Machine and Servo databases discussed in Section 4.1. Setiono performed one 10-fold cross validation evaluation on each of the previously mentioned datasets. The predictive accuracy of NeuroLinear was tested in terms of the generalisation *mean absolute error*:

$$E_{MA} = \frac{\sum_{i=1}^{|P|} |y_i - y_i^{\bullet}|}{|P|}$$

Setiono also provided the average number of rules generated for each dataset.

The second comparison method is a commercial version of the M5 algorithm called Cubist (Quinlan, 1992). Cubist internally utilises model trees with linear regression models. Cubist presents these model trees in the form of a production system. Both Cubist and the GPMCC method were used to perform ten 10-fold cross validation evaluations (equivalent to 100 simulation runs) on the datasets men-

Table 4
Comparison of Cubist and GPMCC

| Dataset | Method | $\overline{GMAE}$ | $\sigma_{GMAE}$ | $\overline{rules}$ | $\sigma_{rules}$ | $\overline{conds}$ | $\sigma_{conds}$ | $\overline{terms}$ | $\sigma_{terms}$ |
|---|---|---|---|---|---|---|---|---|---|
| Elevators | Cubist | 0.0019 | 0.0001 | 18.0400 | 2.3047 | 2.9669 | 0.3206 | 2.9493 | 0.3229 |
| | GPMCC | 0.0018 | 0.0000 | 3.3200 | 0.9522 | 1.7929 | 0.5161 | 8.8217 | 0.8498 |
| House-16H | Cubist | 16355.2840 | 375.0254 | 35.7100 | 4.7637 | 4.4092 | 0.4649 | 4.1570 | 0.4291 |
| | GPMCC | 24269.8000 | 3889.8900 | 5.1300 | 1.2032 | 2.6361 | 0.5451 | 7.2623 | 1.5700 |
| Federal Reserve ED | Cubist | 0.0999 | 0.0137 | 17.0300 | 4.0613 | 2.5765 | 0.3249 | 4.8203 | 0.4984 |
| | GPMCC | 0.1514 | 0.0366 | 2.8700 | 0.6765 | 1.5553 | 0.4031 | 7.6408 | 1.1640 |
| Function Example | Cubist | 0.9165 | 1.4499 | 41.4600 | 5.3756 | 2.8032 | 0.2843 | 1.1384 | 0.1221 |
| | GPMCC | 1.3713 | 1.8050 | 4.1700 | 0.5695 | 2.0916 | 0.2158 | 2.1400 | 0.1654 |
| Lena Image | Cubist | 5.0160 | 0.2102 | 32.8000 | 4.5969 | 4.3619 | 0.4775 | 3.9577 | 0.3999 |
| | GPMCC | 6.4107 | 0.2466 | 6.5500 | 1.6229 | 3.0743 | 0.5332 | 7.4953 | 1.0319 |
| Mono Sample | Cubist | 13.3550 | 0.1749 | 35.2700 | 4.5480 | 3.6664 | 0.3942 | 4.4224 | 0.4533 |
| | GPMCC | 13.7162 | 0.1762 | 4.5300 | 1.1845 | 2.4217 | 0.5613 | 4.7366 | 0.4293 |
| Stereo Sample | Cubist | 11.2470 | 0.1507 | 11.5500 | 2.8298 | 2.9741 | 0.4580 | 4.9500 | 0.5000 |
| | GPMCC | 11.2498 | 0.1923 | 2.6100 | 0.7092 | 1.3883 | 0.4392 | 7.8308 | 0.6162 |
| Time-series | Cubist | 0.7664 | 0.1853 | 30.2400 | 3.8379 | 3.5024 | 0.3778 | 3.1646 | 0.3315 |
| | GPMCC | 0.6442 | 0.3718 | 3.6700 | 0.8996 | 2.0062 | 0.4271 | 5.0386 | 1.6865 |

tioned previously, in order to determine the generalisation mean absolute error. The average number of generated rules, rule conditions (the length of the path from the root to the terminal node) and rule terms (the number of terms in the model) were also obtained for each dataset.

Table 3 shows the results for Cubist, the GPMCC method and NeuroLinear for the five databases mentioned above. In all the cases the GPMCC method was the least accurate of the three methods in terms of generalisation accuracy (but not significantly). Cubist, on the other hand, was the most accurate of the three methods. However, the number of rules generated by the GPMCC method was significantly less than that of the other methods, with the exception of the Servo and Machine datasets. Also, the total complexity of the GPMCC method, in terms of the average number of rules, the average number of rule conditions and the average number of rule terms was significantly less than that of Cubist.

The GPMCC method and Cubist were also compared on the remaining problems. Once again, ten 10-fold cross validation evaluations were performed in order to obtain the generalisation mean absolute error. Additionally, the average number of generated rules, rule conditions and rule terms were obtained for each database.

Table 4 shows the results for Cubist and the GPMCC method for the remaining databases not used in Table 3. In all cases the GPMCC method outperformed Cubist in terms of the average number of rules generated and the total complexity. In fact, the average number of rules generated by the GPMCC method and the total complexity of the GPMCC method were significantly less than that of Cubist. Additionally, the GPMCC method even managed to outperform Cubist in terms of the generalisation mean absolute error on some of the datasets, *i.e.* Elevators and Time-series. However, there is no statistically significant difference in accuracy.

### 4.4. Rule quality

This section discusses the quality of the rules inferred by the GPMCC method for a selection of the datasets in Section 4.1 (the reader is referred to (Potgieter, 2003) for results of the other problems). Each model tree represents the best outcome of ten 10-fold cross validation evaluations in terms of the mean squared error on the generalisation set. The GPMCC method was initialised using Table 2. For the best model trees given below, values between parenthesis show how many patterns are covered by the antecedent of a rule. Values between angle brackets indicate the mean squared error of the rule on patterns covered by the antecedent of that rule. For both types of parenthesis, the first value between parenthesis represents the outcome for the training set and the second value represents the outcome for the validation set.

- The best model tree describing the **Abalone** dataset is as follows:

```
if (Sex == "F") {
  if (Viscera > 0.545792) {
    Rings = 13.6301*pow(Shell,1)
          -16.1805*pow(Viscera,1)
          -26.6776*pow(Shucked,1)
          +13.0827*pow(Whole,1)
          +8.99643;
    //(1, 0) <48.201, 0>
  } else {
    Rings =-44.7887*pow(Shell,1)*pow(Length,1)
          +42.2753*pow(Shell,1)
          +132.724*pow(Viscera,3)*pow(Diameter,2)
          -21.1746*pow(Viscera,1)
          +22.7078*pow(Shucked,2)
          -45.8945*pow(Shucked,1)
          -6.61542*pow(Whole,2)
          +28.4355*pow(Whole,1)
          +0.782952*pow(Height,1)
          +4.46808;
    //(1027, 130) <6.01941, 5.99693>
  }
} else {
  Rings = 13.1915*pow(Shell,1)
        +17.8778*pow(Viscera,1)*pow(Whole,1)*pow(Diameter,1)
        -40.99*pow(Viscera,1)*pow(Length,1)
        +59.2038*pow(Shucked,2)
        -35.4237*pow(Shucked,1)*pow(Whole,1)
        -42.489*pow(Shucked,1)
        +0.10544*pow(Whole,4)
        +27.2256*pow(Whole,1)
        +4.30859*pow(Diameter,1)
        +3.96395;
  //(2313, 288) <3.83083, 3.1415>
}
TMSE: 4.51687
VMSE: 4.02955
GMSE: 4.904
```

  The largest order used by the models of the model tree is 5. Also, the first rule represents an outlier. Obviously this outlier skewed the coefficients of the GPMCC method so drastically that the GPMCC method had no choice but to isolate it. This indicates that this training pattern should be removed from the dataset.
- The best model tree describing the **Auto-mpg** dataset is as follows:

```
if (displacement > 97.2829) {
  mpg = -0.061885*pow(model_year,1)*pow(cylinders,1)
        +0.933105*pow(model_year,1)
        +0.00151765*pow(weight,1)*pow(cylinders,1)
        -0.0147998*pow(weight,1)
        -0.037649*pow(horsepower,1);
  //(251, 30) <7.37801, 7.72467>
} else {
  mpg = -1.18039*pow(origin,1)
        +0.0479332*pow(model_year,2)
        -0.0172612*pow(model_year,1)*pow(cylinders,2)
        -5.99653*pow(model_year,1)
        -0.00545806*pow(weight,1)
        -0.0244613*pow(horsepower,1)
        -0.0546243*pow(displacement,1)
        +14.5401*pow(cylinders,1)
        +192.894;
  //(61, 10) <17.2217, 3.97088>
}
TMSE: 9.30258
VMSE: 6.78622
GMSE: 3.48597
```

  Only two non-linear rules where generated. The largest order utilised by the models of the model tree is 3.
- The best model tree describing the **Elevators** dataset is as follows:

```
if (SaTime1 < -0.000562935) {
  if (diffRollRate > -0.011995) {
    Goal = 0.0848297*pow(Sa,1)*pow(diffClb,1)
           -56.4038*pow(Sa,1)
           -9511.27*pow(SaTime4,2)
           -0.00331258*pow(SaTime3,1)*pow(climbRate,1)
           -1.43111*pow(SaTime1,1)
           +0.747046*pow(diffRollRate,1)
           +0.00236211*pow(absRoll,1)
           +0.00547074*pow(p,1)
           +0.0106479;
    //(2925, 372) <7.95712e-06, 5.63356e-06>
  } else {
    Goal = -36.6557*pow(Sa,1)
           -4101.46*pow(SaTime4,2)
           -0.00368597*pow(SaTime3,1)*pow(climbRate,1)
           +0.0014319*pow(SaTime2,1)*pow(Sgz,1)
           +0.463288*pow(diffRollRate,1)
           +0.00154505*pow(absRoll,1)
           +0.00781973*pow(q,1)
           +0.00305086*pow(p,1)
           +0.0123348;
    //(1189, 151) <3.92677e-06, 3.27794e-06>
  }
} else {
  Goal = -25.0523*pow(Sa,1)
         +0.116456*pow(SaTime4,1)*pow(diffClb,1)
         -0.00669548*pow(SaTime3,1)*pow(climbRate,1)
         +0.433775*pow(diffRollRate,1)
         +0.00126904*pow(absRoll,1)
         +0.00372367*pow(p,1)
         +0.016304;
  //(2886, 353) <4.34178e-06, 4.54399e-06>
}
TMSE: 5.78198e-06
VMSE: 4.78845e-06
GMSE: 4.41977e-06
```

A large number of terms were utilised by the models of the model tree. However, the maximum polynomial order of the models in the model tree is 4.

- The best model tree describing the **Federal Reserve Economic Data** dataset is as follows:

```
if (Y3TCMR > 15.8756) {
  M1CDR = 0.04954*pow(TWEIMC,1)
          +0.00308282*pow(M3TBRAA,2);
  //(3, 1) <0.246856, 0.641734>
} else {
  if (M3TBRSM > 11.1668) {
    M1CDR = -0.132345*pow(TLLACB,1)
            +0.0941262*pow(TCD,1)
            +0.56461*pow(M1MS,1)
            -0.162584*pow(BCACB,1)
            +0.206319*pow(Y3TCMR,1)
            -0.0400925*pow(M3TBRAA,1)
            +0.446434*pow(Y30CMR,1);
    //(82, 8) <0.11477, 0.0222504>
  } else {
    M1CDR = 0.450881*pow(M1MS,1)
            -0.0102812*pow(DDCB,1)
            +0.00126507*pow(CCMS,1)
            +0.0538425*pow(BCACB,1)
            -0.00205841*pow(Y5TCMR,2)
            -0.356311*pow(Y5TCMR,1)
            +0.00877064*pow(Y3TCMR,2)
            +0.00308282*pow(M3TBRAA,2)
            +0.69845*pow(Y30CMR,1)
            -0.11687;
    //(754, 96) <0.0422099, 0.0382938>
  }
}
TMSE: 0.0500334
VMSE: 0.0428185
GMSE: 0.0250136
```

A large number of terms were utilised by the models of the model tree. The maximum polynomial order of the models in the tree is 2.

- The best model tree describing the **Function Example** dataset is as follows:

```
if (type == "A") {
  y = 2.01245*pow(x,1)
      +0.512857;
  //(266, 22) <0.0912585, 0.0873773>
} else {
  if (type == "C") {
    if (x > 0.997578) {
      y = 1.48816*pow(x,2)
          -6.83703*pow(x,1)
          +2.09168;
      //(66, 9) <0.0959942, 0.0494538>
    } else {
      y = 1.01387*pow(x,3)
          +400.504;
      //(204, 31) <0.100298, 0.0883689>
    }
  } else {
    y = -1.9986*pow(x,1)
        +0.512857;
    //(264, 38) <0.0853964, 0.0836496>
  }
}
TMSE: 0.0920197
VMSE: 0.0828551
GMSE: 0.0811045
```

What is interesting to note, is that the model tree is almost identical to the generating function of Section 4.1.

- The best model tree describing the **Housing** dataset is as follows:

```
if (DIS < 1.81274) {
  MEDV = 0.0311044*pow(LSTAT,2)
         -0.0852524*pow(LSTAT,1)*pow(PTRATIO,1)*pow(NOX,1)
         -0.596869*pow(LSTAT,1)
         +0.431583*pow(RAD,1)*pow(CHAS,1)
         -1.35841*pow(DIS,1)
         +0.119659*pow(RM,3)
         -11.7921*pow(RM,1)
         -0.108755*pow(CRIM,1)
         +83.3143;
  //(63, 8) <33.3665, 3.50684>
} else {
  MEDV = -0.767679*pow(LSTAT,1)*pow(NOX,1)
         -0.729239*pow(PTRATIO,1)
         -0.0137382*pow(TAX,1)
         -0.255243*pow(RAD,1)*pow(RM,1)
         +1.71962*pow(RAD,1)
         -0.59144*pow(DIS,1)
         +2.81649*pow(RM,2)
         -30.0403*pow(RM,1)
         +1.5585*pow(CHAS,1)
         +124.502;
  //(341, 43) <10.7152, 7.98617>
}
TMSE: 14.2475
VMSE: 7.28353
GMSE: 6.29143
```

Only two non-linear rules were obtained. The maximum order of the models of the model tree is 3.

- The best model tree describing the **Lena Image** dataset is as follows:

```
if (blend[7] < 203.273) {
  intensity = -0.0123416*pow(across,1)
              +0.00189951*pow(blend[7],2)
              -0.0477566*pow(blend[6],1)
              +0.326583*pow(blend[5],1)
              +0.694096*pow(blend[4],1)
              -0.00194599*pow(blend[3],2)
              +0.22087*pow(blend[3],1)
              -0.132875*pow(blend[2],1)
              -0.0649604*pow(blend[1],1)
              +1.96046;
  //(12328, 1532) <127.918, 120.595>
} else {
  if (blend[3] > 118.928) {
    if (blend[0] > 49.8796) {
      if (across < 55.2729) {
        intensity = 0.507778*pow(blend[7],1)
                    -0.0444378*pow(blend[6],1)
                    +0.354999*pow(blend[5],1)
                    +0.659356*pow(blend[4],1)
                    -0.303265*pow(blend[3],1)
                    -0.176485*pow(blend[2],1);
        //(38, 6) <351.494, 62.1143>
      } else {
        intensity = 0.0105132*pow(blend[7],2)
                    -0.0205424*pow(blend[7],1)*pow(blend[4],1)
                    -0.00172814*pow(blend[5],2)
                    +0.00143276*pow(blend[5],1)*pow(blend[1],1)
                    +0.317081*pow(blend[5],1)
                    +5.42638*pow(blend[4],1)
                    -0.543182*pow(blend[1],1)
                    -0.167443*pow(blend[0],1)
                    -391.867;
        //(734, 100) <150.048, 128.225>
      }
    } else {
      intensity = 1.94436*pow(blend[2],1);
      //(2, 0) <1.63576, 0>
    }
  } else {
    intensity = 1.47793*pow(blend[5],1)
                -0.62556*pow(blend[2],1)
                +0.263644*pow(blend[1],1);
    //(4, 1) <0.00493363, 1451.8>
  }
}
TMSE: 129.747
VMSE: 121.659
GMSE: 106.126
```

Essentially, the rules consist of linear blends of the context pixels to obtain the predicted pixel value. One of the rules represents an outlier. Also, the maximum polynomial order of the models is 2.

- The best model tree describing the **Machine** dataset is as follows:

```
if (CACH < 128.391) {
  if (MMIN < 25112.4) {
    PRP = 0.0347948*pow(CHMAX,1)*pow(CACH,1)
          -0.14693*pow(CHMIN,2)
          +0.00117564*pow(CHMIN,1)*pow(MMIN,1)
          +0.00558463*pow(MMAX,1)
          -0.0010641*pow(MMIN,1);
    //(160, 21) <1212.63, 568.998>
  } else {
    PRP = 31.0917*pow(CHMIN,1)
          +0.00430516*pow(MMIN,1);
    //(1, 0) <127.71, 0>
  }
} else {
  PRP = -2.33197*pow(CHMAX,1)
        +30.9474*pow(CHMIN,1)
        +0.00440527*pow(MMIN,1);
  //(6, 0) <736.413, 0>
}
TMSE: 1189.02
VMSE: 568.998
GMSE: 333.024
```

A small number of linear rules where generated. The largest order utilised by the models of the model tree is 2. Once again one of the rules represents an outlier.

- The best model tree describing the **Mono Sample** dataset is as follows:

```
if (t < 10566.2) {
  if (buf[0] > 130.813) {
    y = 0.869102*pow(buf[2],1)
        -0.398713*pow(buf[1],1)
        +0.436347*pow(buf[0],1)
        +11.8149;
    //(3932, 497) <307.129, 311.468>
  } else {
    y = 0.8834*pow(buf[2],1)
        -0.454296*pow(buf[1],1)
        +0.475617*pow(buf[0],1)
        +12.0682;
    //(4510, 540) <314.362, 334.597>
  }
} else {
  if (t < 20022.7) {
    if (buf[1] < 145.598) {
      y = -0.00243541*pow(buf[2],1)*pow(buf[1],1)
          +1.17106*pow(buf[2],1)
          +0.00179399*pow(buf[1],2)
          -1.04915*pow(buf[1],1)
          +0.721939*pow(buf[0],1)
          +28.1399;
      //(5460, 692) <332.485, 356.925>
    } else {
      y = 0.00263594*pow(buf[2],2)
          -0.447044*pow(buf[1],1)
          +0.566219*pow(buf[0],1)
          +61.1108;
      //(2109, 235) <286.787, 252.748>
    }
  } else {
    y = 1.08554*pow(buf[2],1)
        -0.646192*pow(buf[1],1)
        +0.366292*pow(buf[0],1)
        +24.683;
    //(9495, 1225) <263.164, 249.476>
  }
}
TMSE: 295.787
VMSE: 297.108
GMSE: 294.327
```

A small number of linear rules where generated. The largest order utilised by the models of the model tree is 2.

- The best model tree describing the **Servo** dataset is as follows:

```
if (motor == "D") {
  class = -0.343752*pow(pgain,1)
          +2.13126;
  //(17, 2) <0.267481, 0.999964>
} else {
  if (motor == "E") {
    if (screw == "A") {
      class = -0.0971804*pow(vgain,1)*pow(pgain,1)
              +0.59491*pow(vgain,1)
              -0.489579*pow(pgain,3)
              +7.57494*pow(pgain,2)
              -38.6539*pow(pgain,1)
              +65.5058;
      //(6, 1) <1.43629, 0.00659144>
    } else {
      class = -0.343752*pow(pgain,1)
              +2.13126;
      //(19, 2) <0.126267, 0.00564462>
    }
  } else {
    class = -0.122795*pow(vgain,1)*pow(pgain,1)
            +0.730519*pow(vgain,1)
            -0.447423*pow(pgain,3)
            +7.05858*pow(pgain,2)
            -36.4128*pow(pgain,1)
            +61.5672;
    //(91, 12) <0.39332, 0.016951>
  }
}
TMSE: 0.386136
VMSE: 0.13066
GMSE: 0.0315292
```

A small number of rules where generated, however some of the rules are non-linear. The maximum order of the models of the model tree is 3.

- The best model tree describing the **Stereo Sample** dataset is as follows:

```
if (t > 23500.3) {
  y = 0.510853*pow(left[3],1)
      +0.754001*pow(right[2],1)
      -0.0890549*pow(left[2],1)
      -0.677194*pow(right[1],1)
      +0.333603*pow(left[1],1)
      +0.393169*pow(right[0],1)
      -0.180878*pow(left[0],1)
      -5.84898;
  //(6356, 797) <198.139, 195.359>
} else {
  if (t > 7772.91) {
    if (left[1] < 167.104) {
      y = 0.595663*pow(left[3],1)
          +0.584884*pow(right[2],1)
          -0.14832*pow(left[2],1)
          -0.646768*pow(right[1],1)
          +0.375642*pow(left[1],1)
          +0.477733*pow(right[0],1)
          -0.185438*pow(left[0],1)
          -6.5451;
      //(11317, 1434) <219.241, 215.915>
    } else {
      y = 0.616402*pow(left[3],1)
          +0.603696*pow(right[2],1)
          -0.18247*pow(left[2],1)
          -0.623408*pow(right[1],1)
          +0.39671*pow(left[1],1)
          +0.452291*pow(right[0],1)
          -0.199426*pow(left[0],1)
          -8.13588;
      //(1241, 133) <211.091, 152.021>
    }
  } else {
    y = 0.716837*pow(left[3],1)
        +0.639319*pow(right[2],1)
        -0.394613*pow(left[2],1)
        -0.560577*pow(right[1],1)
        +0.459409*pow(left[1],1)
        +0.430266*pow(right[0],1)
        -0.211955*pow(left[0],1)
        -9.64711;
        //(6230, 779) <173.144, 164.914>
    }
  }
  TMSE: 202.083
  VMSE: 195.358
  GMSE: 189.556
```

Essentially, the rules consist of linear blends of the left and right channels to obtain the predicted sample.

- The best model tree describing the **Time-series** dataset is as follows:

```
if (t2 < -0.0185124) {
  if (t1 < 0.0263429) {
  if (t0 > 0.00343757) {
    y = -2.63082*pow(t0,1)
        +0.972257*pow(t1,1)*pow(t2,1)
        -0.0932611*pow(t2,1)
        +0.326075;
    //(101, 12) <0.123809, 0.0717015>
  } else {
    y = 1.00471*pow(t0,1)*pow(t1,1)*pow(t2,1)
        +0.669822;
    //(97, 8) <0.102582, 0.037943>
  }
 } else {
    y = -0.19592*pow(t1,1)
       +0.979469*pow(t2,1)
       +0.342466;
  //(193, 25) <0.0924138, 0.0926662>
 }
} else {
  y = 0.305796*pow(t1,1)
      -1.39869*pow(t2,2)
      +1.4918;
  //(409, 55) <0.0823577, 0.0943271>
}
TMSE: 0.0924692
VMSE: 0.086686
GMSE: 0.0857686
```

What is interesting to note, is that the model tree is almost identical to the generating function of Section 4.1.

For a large proportion of the above solutions, the utilised polynomial order was no greater than 3. This indicates that cubic surfaces sufficiently describe most databases, including time-series.

For some of the above results, outliers in the dataset were detected and isolated by rules. This indicates that there is still redundancy to be removed from the model tree solutions. Also, the removal of these outliers will improve the generalisation accuracy of the models. These outliers should be removed by some heuristic, *e.g.* rules that cover a smaller number of patterns than some threshold should be removed and the patterns covered by the rules should be discarded from the training set. Thus, further improvements in accuracy and complexity are possible.

## 5. Conclusion

This paper presented a genetic program for mining continuous-valued classes (GPMCC). The performance of the GPMCC was evaluated against other algorithms such as Cubist and NeuroLinear for a wide variety of problems. Although the generalisation ability of the GPMCC method was slightly worse than the other methods, the complexity and number of generated rules were significantly smaller than that of other methods. The GPMCC method was also fairly robust, in that the parameter choices did not significantly effect any outcomes of the GPMCC method. The success of the GPMCC method can be attributed to the specialized mutation and crossover operators, and can also be attributed to data clustering. Another important aspect to the GPMCC method is the development of a fragment pool, which served as a belief space for the genetic program. The fragments of the fragment pool resulted in structurally optimal models for the terminal nodes of the GPMCC method. The fitness function was also crucial, because it penalised chromosomes with a high level of complexity.

Although the genetic program presented in this chapter seems to be fairly effective both in terms of rule accuracy and complexity, the algorithm was not particularly fast. The speed of the algorithm is seriously affected by the recursive procedures used to perform fitness evaluation, crossover and mutation on the chromosomes (model trees). This problem can be solved in two ways: implement a model tree as an array or change the model tree representation to a production system.

If the model trees of the genetic program are represented as an array, clever indexing of the array will negate the need for any recursive functions. However, the array would have to represent a full binary tree which could unnecessarily waste system memory if the model trees are sparse. If the model tree representation is changed to a production system, the mutation and crossover operators will have to be re-investigated.

Envisioned future developments to the GPMCC method include the revision of the attribute tests. These attribute tests could be revised to implement non-linear separation boundaries between continuous classes. The GASOPE method could then be used to efficiently approximate these non-linear separation boundaries. The fragment pool discussed in this chapter could be used to implement a function set for the separation boundaries, in a manner similar to that of the terminal set.

## References


Abass, H., Saker, R., & Newton, C. (Eds.). (2002). *Data mining: A heuristic approach*. Idea Publishing Group.

Bäck, T., Fogel, D., & Michalewicz, T. (Eds.). (2000a). *Evolutionary computation 1: Advanced algorithms and operators*. IOP Press.

Bäck, T., Fogel, D., & Michalewicz, T. (Eds.). (2000b). *Evolutionary computation 1: Basic algorithms and operators*. IOP Press.

Bishop, C. (1992). *Neural networks for pattern recognition*. Oxford University Press.

Blake, C., Keogh, E., & Merz, C. (1998). UCI repository of machine learning databases. Department of Information and Computer Science, University of California Irvine. http://www.ics.uci.edu/~mlearnMLRepository.

Burden, R., & Faires, J. (1997). *Numerical analysis* (6th ed.). Brooks/Cole Publishing Company.

Clark, P., & Boswell, R. (1991). Rule induction with CN2: Some recent improvements. In *Proceedings of the fifth European working session on learning* (pp. 151–163). Berlin: Springer.

Clark, P., & Niblitt, T. (1989). The CN2 induction algorithm. *Machine Learning, 3*, 261–283.

Dennis, J., Jr., & Schnabel, R. (1983). *Numerical methods for unconstrained optimization and nonlinear equations*. Englewood Cliffs, NJ: Prentice-Hall.

Eggermont, J., Eiben, A., & van Hemert, J. (1999). Adapting the fitness function in GP for data mining. In R. Poli, P. Nordin, W. Langdon, & T. Fogarty (Eds.), *Genetic programming. Proceedings of EuroGP'99, 26–27 1999* (Vol. 1598, pp. 193–202). Goteborg, Sweden: Springer-Verlag [Online]. Available fromciteseer.nj.nec.com/eggermont99adapting.html.

Engelbrecht, A., & Brits, R. (2002). Supervised training using an unsupervised approach to active learning. In *Neural processing letters* (pp. 247–269). Netherlands: Kluwer Academic Publishers.

Engelbrecht, A., Schoeman, L., & Rouwhorst, S. (2001). A building-block approach to genetic programming for rule discovery. In C. N. H. A. Abbass & R. A. Sarker (Eds.), *Data mining: A heuristic approach* (pp. 174–190). Idea Group Publishing.

Forgy, E. (1965). Cluster analysis of multivariate data: Efficiency versus interpretability of classification. *Biometrics, 21*, 768–769.

Freitas, A. (1997). A genetic programming framework for two data mining tasks: Classification and generalized rule induction. In J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, & R. Riolo (Eds.), *Genetic programming 1997: Proceedings of the second annual conference, Stanford University, 13–16 1997* (pp. 96–101). Morgan Kaufmann [Online]. Available fromciteseer.nj.nec.com/43454.html .

Fu, L. (1994). Rule generation from neural networks. *IEEE Transactions on Systems, Man and Cybernetics, 24*(8), 1114–1124.

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Computation, 4*, 1–58.

Geom, Y. (1999). Genetic recursive regression for modelling and forecasting real-world chaotic time series. *Advances in Genetic Programming, 3*, 401–423.

Goldberg, G., & Deb, K. (1991). A comparitive analysis of selection schemes used in genetic algorithms. In G. Rawlins (Ed.), *Foundations of genetic algorithms* (pp. 69–93). Morgan-Kaufman.

Kaboudan, M. (1999). Genetic evolution of regression models for business and economic forecasting. *Proceedings of the congress on evolutionary computation* (vol. 2, pp. 1260–1268). IEEE Press.

Koza, J. (1992). *Genetic programming: On the programming of computers by means of natural selection*. MIT Press.

Marmelstein, R., & Lamont, G. (1998). Pattern classification using a hybrid genetic program-decision tree approach. In J. Koza (Ed.), *Genetic programming 98: Proceedings of the third international conference* (pp. 223–231). Morgan Kaufman.

Poli, R., & Langdon, W. (2002). *Foundations of genetic programming*. Springer-Verlag.

Potgieter, G. (2003). Mining continuous classes using evolutionary computing. Master's thesis. Department of Computer Science, University of Pretoria.

Potgieter, G., & Engelbrecht, A. (2002). Structural optimization of learned polynomial expressions using genetic algorithms. In *Proceedings of the 4th Asia-Pacific conference on simulated evolution and learning*.

Potgieter, G., & Engelbrecht, A. (2007). Genetic algorithms for the structural optimisation of learned polynomial expressions. *Applied Mathematics and Computation, 186*, 1441–1466.

Quinlan, J. (1983). Learning efficient classification procedures and their application to chess endgames. In R. Michalski, J. Carbonell, &

Mitchell (1995). *Machine learning: An artificial intelligence approach* (vol. 1, pp. 463–482). Palo Alto: Tioga Press.

Quinlan, J. (1992). Learning with continuous classes. In Adams & Sterling (Eds.), *Proceedings of artificial intelligence'92* (pp. 343–348). Singapore: World Scientific.

Quinlan, J. (1993). *C4.5: Programs for machine learning*. San Mateo: Morgan Kaufman.

Salomon, D. (2000). *Data compression* (2nd ed.). Springer.

Setiono, R. (2001). Generating linear regression rules from neural networks using local least squares approximation. In J. Mira & A. Prieto (Eds.). *Connectionist model of neurons, learning processes, and artificial intelligence, Proceedings of the 6th international work-conference on artificial and natural neural Networks, June* (Vol. 1, pp. 277–284). Granada, Spain: Springer.

Setiono, R., & Hui, L. (1995). Use of quasi-Newton method in a feedforward neural network construction algorithm. *IEEE Transactions on Neural Networks, 6*(1), 273–277.

Setiono, R., & Leow, W. (2000). Pruned neural networks for regression. In R. Mizoguchi & J. Staney (Eds.), *Proceedings of the 6th Pacific Rim conference on artificial intelligence, PRICAI 2000. Lecture Notes in AI* (Vol. 1886, pp. 500–509). Melbourne, Australia: Springer.

Setiono, R., Leow, W., & Zurada, J. (2002). Extraction of rules from artificial neural networks for nonlinear regression. *IEEE Transactions on Neural Networks, 13*(3), 564–577.

Towell, C., & Shavlik, J. (1994). Refining symbolic knowledge using neural networks. *Machine Learning, 12*, 321–331.

Viktor, H., Engelbrecht, A., & Cloete, I. (1995). Reduction of symbolic rules from neural networks using sensitivity analysis. In *Proceedings of the IEEE international joint conference on neural networks, Perth, Australia* (pp. 1022–1026).

Zurada, J. M. (1992). *Introduction to artificial neural systems*. PWS Publishing Company.