Genetic Programming for Data Classification: Refining the Search Space

J. Eggermont J.N. Kok W.A. Kosters

Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden
The Netherlands

Abstract

When using Genetic Programming to evolve decision trees for data classification the search space sizes tend to become extremely large and sometimes theoretically infinite in size, depending on the representation. In this paper we present a method derived from the well-known decision tree construction algorithm C4.5 to refine and thereby reduce the search space sizes for our decision tree evolvers. We will show that using this refinement method we can significantly improve classification performance.

1 Introduction

Data classification is a particularly difficult problem for evolutionary algorithms mainly because of two reasons. First of all evolutionary algorithms for data classification have to compete with deterministic algorithms like C4.5 [8] which are faster since they only have to make a small number of passes through the training-set. Secondly, unlike machine learning and in particular neural network approaches, evolutionary algorithms search for a classifier (usually decision trees) while algorithms like C4.5 and Back-Propagation construct or build a classifier based on heuristics, rules or mathematical models. The size of the spaces in which the evolutionary algorithms are searching is usually very large and sometimes even infinite (at least theoretically). Unlike other optimization problems, there is no representation for all possible solutions (decision trees) but rather the final solution quality is partially dependent on the chosen representation, the so-called language bias. Thus instead of searching in the space of all possible classifiers we can only search in a restricted space. However, this does not mean that this search space is by any means small as we will show for different data sets. In this paper we use tree-based Genetic Programming (GP) [1, 7] to evolve decision trees for binary classification problems.

The outline of this paper is as follows. We start by describing the decision tree format that we will be using. Next we show how to calculate the search space size given a set of possible internal and external nodes. We then define three decision tree representations by specifying the set of possible internal and external nodes that can occur in our trees. In Section 4 we describe our experiments followed by the results. Finally, in the last section we draw some conclusions and look ahead towards future research.

2 Full Atomic Representations

In this paper we use *full atomic* representations. Unlike the *atomic* representation used in [3] which only uses atoms in its terminal set and Boolean operators as internal nodes, a *full atomic* representation also uses atoms in the internal nodes. Each atom is syntactically a predicate of the form (*attribute operator value*(s)), where *operator* is a function returning a Boolean value (e.g., $<$, $>$ or $=$). In the leaf nodes we have *class assignment* atoms of the form ($class := C$), where C is a category selected from the domain of the attribute to be predicted. A small example tree can be seen in Figure 1. A *full atomic* tree classifies an instance I by traversing the tree from root to leaf node. In each non-leaf node an atom is evaluated. If the result is true the right branch is traversed, else the left branch is taken. This is done for all internal nodes until a leaf node containing a *class assignment* node is reached, resulting in the classification of the instance. The advantage of this representation is two-fold. First of all unlike an *atomic* decision tree, we do not have to traverse the entire tree to classify a data record but only a single path from the root of the tree to a leaf (*class assignment*) node. This can potentially save a lot of time. Secondly, the *atomic* representation was limited to binary classifications while a *full atomic* representation can potentially be used for any n-ary classification problem.

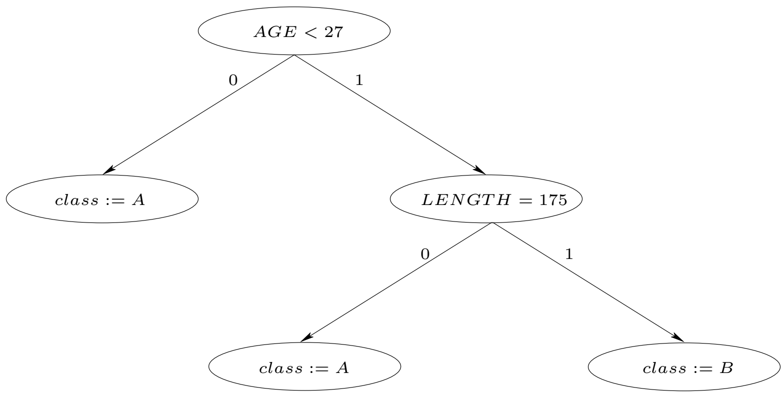

Figure 1: An example of a *full atomic* tree

2.1 Calculating the size of the search space

Since every decision tree using a *full atomic* representation is also a full binary tree we can calculate the size of the search space for each specific *full atomic* representation and data set. In order to calculate the size of the search space of

GP algorithms using a *full atomic* representation for a given data set we use some formulae from discrete mathematics.

Let N be the number of tree nodes. The total number of binary trees with N nodes b_N can be calculated using the Catalan number

$$b_N = \frac{1}{N+1}\binom{2N}{N}. \tag{1}$$

Let n be the number of internal tree nodes. The total number of tree nodes N in a full binary tree with n internal tree nodes is:

$$N = 2n + 1. \tag{2}$$

We can now combine these two equations into the following lemma:

Lemma 2.1 *The total number of full binary trees with $2n+1$ nodes is $\frac{1}{n+1}\binom{2n}{n}$.*

Proof Let B be a tree with n nodes. In order to transform this tree into a full binary tree with $2n+1$ nodes we need to add $n+1$ nodes. This can only be done in one way. $\square$

Since in a *full atomic* tree the contents of a node is dependent on the set of internal nodes and the set of external nodes we can define the total number of *full atomic* trees with a maximum tree size of N nodes, a set of internal nodes I and a set of terminal nodes T as follows.

Lemma 2.2 *The total number of full atomic trees with at most N nodes (N odd), a set of internal nodes I and a set of terminal nodes T is*

$$\sum_{n=1}^{\frac{N-1}{2}} \frac{1}{n+1}\binom{2n}{n} \times |I|^n \times |T|^{n+1}$$

Note that — contrary to decision trees in algorithms such as C4.5 — the same atom can occur more than once in a path from root to leaf node.

We will use these formulae later to calculate the sizes of the search spaces in our experiments.

2.2 A Simple Representation

By using a *full atomic* representation we have defined the basic shape of our decision trees. We can define the precise decision tree representation by specifying which atoms are to be used. In this section we will specify a simple, but powerful, decision tree representation that uses different types of atoms based on the data type of an atom's *attribute*. For non-numerical attributes we use atoms of the form ($variable_i = value$) for each possible *attribute-value* combination found in the data set. For numerical attributes we define the less-than operator ($<$). Again we use atoms for each possible *attribute-value* combination found in the data set. The idea in this approach is that the GP algorithm will be able to decide the best *value* at a given point in a tree. This *simple* representation is similar to the representation used by Rouwhorst and Engelbrecht [9].

2.3 Refined Representations

One of the drawbacks of the *simple full atomic* representation is the fact that it creates an internal node for each combination of a numerical valued attribute and value that occurs in the data set. In order to reduce the huge search space size that results from the large number of possible internal nodes we will borrow some ideas from C4.5. Decision tree constructing algorithms like C4.5 tend to be greedy and non-backtracking in nature since finding the smallest decision tree consistent with a specific training-set is NP-complete [5]. In the case of C4.5 and its predecessor ID3 the criteria used were *gain_ratio* and *gain* [8]. These criteria measure the amount of entropy based information gained by splitting a data set on some test. Just like C4.5, our *refined representations* select a single threshold value for splitting a data set on a continuous valued attribute. In effect, from all of the possible internal nodes created by the *simple* representation only one is selected for each numerical valued attribute. Thus for every numerical valued attribute A_i in a data set T we will calculate the *gain* or *gain_ratio* measure for every possible threshold value v_j from the domain D_i of A_i. We will then select for every attribute A_i a threshold value v_t such that $gain(\_ratio)(A_i < v_t) \geq gain(\_ratio)(A_i < v_j)$ *for all* $v_j \in D_i$. We do not expect to see much difference between using the *gain* and *gain_ratio* criteria as we only use them to split the domain of numerical valued attributes into two, but we will try them both to see if there is a significant difference between the two criteria.

2.4 Example

Let us examine the example data set T in Table 1.

Table 1: An example data set

A	**B**	**class**
1	a	yes
2	b	yes
3	c	no
4	d	no

In the case of the *simple* representation we get the following atoms:

- Since attribute **A** has four possible values {1,2,3,4} and is numerical valued we use the $<$ operator: ($\mathbf{A} < 1$), ($\mathbf{A} < 2$), ($\mathbf{A} < 3$) and ($\mathbf{A} < 4$).
- Attribute **B** is non-numerical and thus we use the $=$ operator: ($\mathbf{B} = a$), ($\mathbf{B} = b$), ($\mathbf{B} = c$) and ($\mathbf{B} = d$).

- Finally for the target class we have two terminal nodes: (*class* := *yes*) and (*class* := *no*).

In the case of the *refined* representations we would have the same atoms for attribute **B** and the target classes. Since attribute **A** is numerical valued we will have to determine a threshold value. Using either the *gain* or *gain_ratio* criteria, ($\mathbf{A} < 3$) would be chosen as the possible internal node as it results in the highest *gain*, and *gain_ratio*.

In the case of the *simple* representation we can calculate the size of the search space to be approximately 3×10^{53}, with our default maximum tree size of 63 nodes. In the case of a *refined* representation we have 5 possible internal nodes and 2 possible class assignment nodes. Given the same maximum tree size of 63 nodes and using Lemma 2.2 this would imply a search space size of approximately 1×10^{47}.

3 Multi-layered Fitness

Although we will compare our *full atomic* GP algorithms to other data classification algorithms based on their classification performance, there is a second objective for our *full atomic* GPs which is also very important: understandability. The *simple* and *refined* representations introduced in the previous section are similar to the decision trees constructed by C4.5 and quite easy for humans to understand. However, even the most understandable decision tree representation can result in incomprehensible trees if the trees become too large. In order to keep the size of our decision trees limited we use two methods. The first method is a built in system which prunes decision trees that have more than a pre-determined number of nodes. The second method is the use of a *multi-layered* fitness.

A multi-layered fitness is a fitness which consists of several fitness measures or objectives which are ranked according to their importance. In the case of our *simple* representation we use a multi-layered fitness consisting of two fitness measures that we want to minimize. The primary, and most important, fitness measure is the misclassification percentage. The secondary fitness measure is the number of tree nodes. When the fitness of two individuals is to be compared we first look at the primary fitness. If both individuals have the same misclassification percentage we compare the secondary fitness measures.

4 Experiments and Results

We will compare our *full atomic* GP representations to C4.5 using several data sets from the UCI machine learning data set repository [2]. An overview of the data sets can be seen in Table 2. More details and comparisons to other evolutionary and non-evolutionary algorithms can be found in the forthcoming PhD thesis of the first author.

Each algorithm is evaluated using n-fold cross-validation and the performance is the average misclassification error over n folds. In n-fold cross-validation the

Table 2: An overview of the data sets used in the experiments

data set	records	attributes	classes
Australian credit (statlog)	690	14	2
German credit (statlog)	1000	23	2
Pima Indians diabetes	768	8	2
Heart disease (statlog)	270	13	2
Ionosphere	351	34	2

total data set is divided into n parts. Each part is chosen once as the test set while the other $n-1$ parts form the training set. In all our experiments we used a 10-fold cross-validation.

We will mention the results of C4.5 as reported by Freund and Shapire [4], in order to compare our results to a non-evolutionary decision tree algorithm.

A single GP implementation was used for both *simple* and *refined* representations. It was programmed using the *Evolving Objects* library (EOlib) [6]. EOlib is an Open Source C++ library for all forms of evolutionary computation and is available from `http://eodev.sourceforge.net`.

In our GP system we use the standard GP mutation and recombination operators for trees. The mutation operator replaces a subtree with a randomly created subtree and the crossover operator exchanges subtrees between two individuals. Both the mutation rate and crossover rate are set to 0.9. The population was initialized using the ramped half-and-half initialization [7, 1] method to create a combination of full and non-full trees with a maximum tree depth of 6. We used a generational model (comma strategy) with population size of 100, an offspring size of 200 and a maximum of 99 generations. Parents were chosen by using (5-)tournament selection. We did not use elitism as the best individual was stored outside the population. Each newly created individual, whether through initialization or recombination, was automatically pruned to a maximum number of 63 nodes.

Table 3: The approximate search space sizes for our algorithms

data set	*simple* GP	*refined* GP
Australian Credit	4×10^{120}	1×10^{61}
German Credit	7×10^{100}	5×10^{67}
Pima Indians Diabetes	3×10^{121}	3×10^{53}
Heart Disease	4×10^{105}	1×10^{60}
Ionosphere Disease	5×10^{146}	9×10^{72}

The search space sizes for our algorithms as calculated using Lemma 2.2 are displayed in Table 3. As can be seen there is a huge difference in search space sizes as the sizes for our *simple* GP algorithm are sometimes more than squared the size for our *refined* GP algorithms.

Table 4: The average misclassification rates (in %)

		Simple GP	*Refined* GP		*C4.5*
data set			*gain*	*gain_ratio*	
Australian Credit *	average	22.0	**14.3**	15.4	15.9
	s.d.	3.0	0.5	0.4	
	best	17.0	13.6	14.9	
	worst	25.7	15.5	16.1	
German Credit *	average	**27.1**	28.0	28.6	27.2
	s.d.	0.7	0.7	0.5	
	best	26.1	26.9	27.4	
	worst	28.2	28.8	29.1	
Pima Indians Diabetes	average	**26.3**	26.8	27.6	28.4
	s.d.	1.1	0.5	0.0	
	best	24.3	26.0	27.6	
	worst	28.5	27.7	27.6	
Heart Disease *	average	25.2	**19.8**	20.2	22.2
	s.d.	2.3	2.0	1.1	
	best	22.6	15.9	18.1	
	worst	31.1	23.0	21.5	
Ionosphere Disease	average	12.4	**7.6**	7.6	8.9
	s.d.	1.8	0.7	0.9	
	best	8.0	6.5	6.3	
	worst	14.3	8.5	9.1	

We performed 10 independent runs for our two GP algorithms to obtain the results presented in Table 4. When available from the literature the results of C4.5 are reported. For three data sets (Australian credit, Heart disease and German credit) no results were reported for C4.5. In those three instances, marked with a $*$, we applied C4.5 to the data set ourselves. The best results for each data set are printed in bold.

When we look at the results in Table 4 we see that on three of the five data sets our *refined* GP algorithm using the *gain* criterion performs a lot better than our *simple* GP algorithm and C4.5. If we look at the other two data set sets (German Credit and Pima Indians Diabetes), we see these are also the data sets on which our *simple* GP performs better than or close to C4.5. It is at the moment unclear why on these two data sets the results differ from the other three data sets. If we compare the two *refined* GP algorithms we see that using the *gain* criterion results in better classification results on all five data sets. This is somewhat unexpected as C4.5, which uses the *gain_ratio* criterion, is generally considered to be superior to ID3 which uses the *gain* criterion.

Due to the high number of parameters involved it is difficult to draw hard conclusions. *Refined* GP seems to be the overall winner: it is either much better or only slightly worse than the other methods.

5 Conclusions and Future Research

In this paper we have shown how the *gain* and *gain_ratio* criteria from ID3 and C4.5 can be used to refine the search space sizes for GP algorithms. When successful the refinement (and reduction) of the search space helps our GP algorithms to find better (local) optima. Unfortunately, the *gain* and *gain_ratio* criteria do not always help us determine the optimal threshold values.

At the moment our *refined* GP algorithms only select a single threshold value for each numerical valued attribute and thereby split the domain of such an attribute into two. However, the *gain* and *gain_ratio* criteria can also be used to split the domain of numerical valued attribute into n parts (using $n-1$ threshold values). Hopefully extending our *refined* GP algorithms in this way will improve classification performance on all data sets. The *gain_ratio* criterion should be especially useful as it was designed to find a balance between information gained by splitting a data set into a large number of data subsets and limiting the number of subsets.

References

[1] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming: An Introduction.* Morgan Kaufmann, 1998.

[2] C.L. Blake and C.J. Merz. *UCI Repository of machine learning databases.* http://www.ics.uci.edu/~mlearn/MLRepository.html.

[3] J. Eggermont, A.E. Eiben, and J.I. van Hemert. Adapting the fitness function in GP for data mining. In R. Poli, P. Nordin, W.B. Langdon, and T.C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 195–204. Springer-Verlag, 1999.

[4] Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Proc. 13th International Conference on Machine Learning*, pages 148–146. Morgan Kaufmann, 1996.

[5] L. Hyafil and R.L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5:15–17, 1976.

[6] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: a general purpose evolutionary computation library. In P. Collet et al., editor, *Proceedings of Evolution Artificielle'01*, LNCS. Springer Verlag, 2001. To appear.

[7] J.R. Koza. *Genetic Programming.* MIT Press, 1992.

[8] J.R. Quinlan. *Programs for machine learning.* Morgan Kaufmann, 1993.

[9] S. E. Rouwhorst and A. P. Engelbrecht. Searching the forest: Using decision trees as building blocks for evolutionary search in classification databases. In *Proc. of the 2000 Congress on Evolutionary Computation*, pages 633–638. IEEE Service Center, 2000.