Logical Decision Rules: Teaching C4.5 to Speak Prolog

Kamran Karimi and Howard J. Hamilton

Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2
{karimi,hamilton}@cs.uregina.ca

Abstract. It is desirable to automatically learn the effects of actions in an unknown environment. C4.5 has been used to discover associations, and it can also be used to find causal rules. Its output consists of rules that predict the value of a decision attribute using some condition attributes. Integrating C4.5's results in other applications usually requires spending some effort in translating them into a suitable format. We have modified standard C4.5 so it will optionally generate its rules in Prolog. We have tried to retain as much information as possible in the conversion process. For example, it is possible for the prolog statements to optionally include the certainty values that C4.5 computes for its rules. This is achieved by simulating the certainty values as the probability that the statement will fail no matter what the values of its arguments. Prolog can move from statement to statement and find a series of rules that have to be fired to get from a set of premises to a desired result. We briefly mention how, when dealing with temporal data, the Prolog statements can be used for recursive searches

1 Introduction

This paper describes how C4.5 [4] has been modified to generate Prolog output. C4.5 allows us to extract classification rules from observations of a system. The input to C4.5 consists of a set of records. Each record contains some condition attributes and a single decision attribute. A domain expert should choose the decide attribute as the variable that depends on the others. Though C4.5 has been traditionally used as a classifier, it can also be used to find temporal relations [3].

C4.5 uses a greedy algorithm with one look-ahead step. It computes the information contents of each condition attribute, and the results are used to prune the condition attributes and create classification rules. The output of C4.5 consists of simple predicates such as **if** {(a = 1) AND (b = 2)} **then** {(c = 4)} [83.2%]. An certainty value is assigned to each rule, which determines the confidence in that rule. In the previous example the certainty value is 83.2%. C4.5 creates a decision tree first and then derives decision rules from that tree. After the rules have been created a program in the C4.5 package called "consult" can be used to actually execute the rules. It prompts for the condition attributes and then outputs the appropriate value of the decision attribute. Any other use of the generated rules, including the integration of C4.5's results in other systems, may require some effort in changing the format of the results in order to make them compatible with the input of the other systems.

C4.5's output can be expressed as horn clauses, so we modified C4.5 to directly generate Prolog statements in addition to its normal output. This new output format enables the results to be readily used in any Prolog-based system, thus making them readily machine-processable. Much of the ground work for this has been presented in detail in [2], where we explain C4.5's usability in deriving association and causal rules in a simple environment. In that paper we also showed the usefulness of having Prolog statements in applications beyond those ordinarily expected from C4.5's output, which are usually meant to be used by people instead of automatic machine processing. In this paper we discuss the improvements and modifications required to make the Prolog statements better represent C4.5's results. These include preserving more information in the output and also handling the certainty values assigned by C4.5 to its output rules. This is important because Prolog by default considers all its statements to be always reliable.

The rest of the paper is organized as follows. In Section 2 we explain how C4.5's rules are converted into Prolog Statements. Section 3 describes how certainty information are included in Prolog statements. Section 4 concludes the paper.

2 C4.5 Speaks Prolog

The rules created by C4.5 can easily be represented as Prolog statements. The "c4.5rules" program in the C4.5 package generates rules from a decision tree. We have modified this program to optionally generate its rules in Prolog [2]. In this paper we refer to this modified program as c4.5rules-p. This integration removes the need of a separate pass for the translation of the rules. When the user gives the command line option of '-p 0', a <file stem>.pl file will be created in addition to the normal output. The generated Prolog statements are in the Edinburgh dialect and can be fed to most Prolog interpreters without changes.

We used the c4.5rules-p program to create Prolog statements from data generated by artificial robots that move around in a two dimensional artificial world called URAL [6]. URAL is a typical Artificial Life simulator. The aim of the robot in this artificial world is to move around and find food. At each time-step, the robot randomly chooses to move from its current position to either Up, Down, Left, or Right. The results of none of these actions are known. The robot can not get out of the board, or go through the obstacles that are placed on the board by the simulator. In such cases, a move action will not change the robot's position. The robot can sense which action it takes in each situation.

The goal is to learn the effects of its actions after performing a series of random moves. To do this we save the robot's current situation after each action has taken place.

The simulator records the current *x* and *y* locations, the move direction, and the next value of *x* or *y*. When these data are fed to C4.5, it correctly determines that the next *x* or *y* values depend only on the previous location and the movement direction [2]. One example is a rule like: **if** {(X1 = 1) AND (A1 = L)} **then** (X2 = 0) which states that the result of a move to the left is a decrease in the value of *x*. *A1* and *X1* are the previous action and *x* value respectively. *X2* represents the next *x* value. Table 1 shows a few of the Prolog statements generated by c4.5rules-p from the records of 1000 random movements. The first statement corresponds to the example C4.5 rule just presented. *A1* and *X1* are as explained above, and the classification is done on the next value of *x*.

class(A1, X1, 0) :- A1 = 2, X1 = 1.
class(A1, X1, 2) :- A1 = 3, X1 = 1.
class(A1, X1, 3) :- A1 = 3, X1 = 2.

Table 1. Three Sample Prolog statements generated by C4.5

In Table 1 a value of 2 or 3 for action *A1* could mean going to the left or right, respectively. The output format is as described in [2]. Following C4.5's terminology, the results are designated by a predicate called "class." To merge the Prolog statements generated for different decision attributes (maybe because we need to work with more than one decision attribute) then we could manually rename "class" to something like "classx" and remove any name clashes. In the above case this could happen we want to represent the rules for the moves along the *y* axis too. In the left-hand side of the Prolog statements the condition attributes that are involved in the decision making process come first, and the value of the decision attribute comes last. In the head of the rules, the condition attributes' values are used for the decision making process. All other condition attributes are ignored.

The automatically generated Prolog statements use Prolog's unification operator (=) instead of the comparison operator (=:=). Suppose variable *A* has not been unified before (has no value), then *A* = 3 will cause *A* to take on the value 3, and the statement succeeds. If *A* has already been unified to a number, then an actual comparison will take place and the result will determine whether the statement succeeds or fails, The =:= operator will fail if its operands are not unified. This unification ability allows the user to traverse the rules backward and go from the decision attribute to the condition attributes, or from a set of decision and condition attributes, to the remaining condition attributes. Some example queries are class(A1, 1, 2) (which actions take the creature from *x* = 1 to *x* = 2?) or class(A1, X1, 3) (which action/location pairs immediately lead to *x* = 3?). The ability to interpret the rules in different directions makes C4.5’s discovered rules more useful when represented as Prolog statements.

C4.5rules can generate rules that rely on threshold testing and set membership testing. If we use the standard Prolog operators of =< and > for threshold testing, and implement a simple member() function for testing set membership, then we would not be able to traverse the rules backward, as they lack the ability to unify variables. So if we had a clause like: class(A, B, C) :- A =< B, member(A, C), then we would be unable to use a query like class(A, 3, [1, 2, 3]), because *A* is not unified, and Prolog can not perform the test =< on variables that are not unified. Adding the unification ability to =<, > and member() will remove this limitation. For example, a unifying $X > 10$ would choose a value above 10 for *X* if it is not already unified, and a unifying member(*X*, [1, 2, 3]) would unify *X* with one of 1, 2, or 3 if it is not already unified. Both cases would always succeed if *X* is not unified, because they make sure that the variable is unified with an appropriate value, but could fail if *X* is already unified because in that case a test will be performed.

In [2] we provide simple code to do the unification. This unification ability allows Prolog statements to be traversed backwards, and so they can be used for automatic planning purposes. The planning ability was achieved by first noticing that in Table 1 we are dealing with temporal data, since the decision attribute (the next *x* position) is actually the same as one of the condition attributes seen at a later time. This property can be used to search for a sequence of moves, each of which changes either the *x* or *y* position by one and get to a desired destination. Suppose we are at $x = 1$ and the aim is to go from there to $x = 3$. From Table 1 Prolog can conclude that to go to $x = 3$ one has to be at $x = 2$ and perform a Right move, but to be at $x = 2$, one has to be at $x = 1$ and again go to the right. This complete the planning because we are already at $x = 1$. In other words, $X1 = 1$ in a Prolog statement actually means class(_, _, 1), so the third statement in Table 1 can be converted to this: class(A1, X1, 3) :- A1 = 3, X1 = 2, class(_, _, 2). Intuitively this means that to go to $x = 3$ with a move to the right, one has to be at $x = 2$ (class(_, _, 2)), and we do not care how we got there. Now prolog will have to satisfy class(_, _, 2) by going to $x = 2$. The changes that have to be made to the statements to allow Prolog to do such recursive searches come in detail in [2].

Nonetheless, the representation of the rules in [2] is inadequate for cases where only some of the condition attributes are used for classification. We use example Prolog statements from the more complex Letter Recognition Database from University of California at Irvine's Machine Learning Repository [1] to illustrate this inadequacy. This database consists of 20,000 records that use 16 condition attributes to classify the 26 letters of the English alphabet. The implementation in [2] gives us the kind of rules seen in Table 2 below.

class(A10, A12, A13, A14, A15, 8) :- A10 = 8, A12 = 5, A13 = 3, A14 = 8, A15 = 5.
class(A7, A13, A14, A16, 8) :- A7 = 9, A13 = 0, A14 = 9, A16 = 7.
class(A6, A10, A11, A13, 8) :- A6 = 9, A10 = 7, A11 = 6, A13 = 0.

Table 2. Some Prolog statements generated from the Letter Recognition Database.

The 16 attributes are named A1 to A16. The decision attribute encodes the index of the letters. Table 2 shows some of the rules created for the letter "I." The problem with this form of rule output is that Prolog's variable names are limited in scope to the statement in which they appear. In the last two statements, for example, as far as Prolog is concerned A6 and A7 represent the same thing: Both are place-holders for the first argument of the predicate "class" that has a total of five arguments. This limited name scope means that the user can not use the name of a variable as he knows it to get its value. A representation like this would allow the user to perform queries such as class(9, 7, A14, 0, 8), and get the answer A14 = 6, which is probably not what he had in mind. This happens because Prolog is using the last statement to derive this result, and that statement actually concerns A11 and not A14.

To prevent this loss of information and the subsequent confusion, the modified c4.5rules program was changed to preserve all condition attributes in the left-hand side of the Prolog statements. This change allows Prolog to distinguish among the condition attributes by using their position, so now there is a way for the user to specify the variables unambiguously. The resulting statements are shown in Table 3.

class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, 8) :- A10 = 8, A12 = 5, A13 = 3, A14 = 8,
class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, 8) :- A7 = 9, A13 = 0, A14 = 9, A16 = 7.
class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, 8) :- A6 = 9, A10 = 7, A11 = 6, A13 = 0.

Table 3. Prolog statements with all condition attributes present.

The statements are longer, but now users can make sure Prolog understands what a query means. They can issue a query such as class(_, _, _, _, _, _, 9, _, _, _, _, _, 0, A14, _, 7, 8) and get the correct result of A14 = 9.

3 Translating Uncertain Rules to Prolog

C4.5 assigns a certainty value to each rule it generates, which shows how reliable that rule is. This value is printed to the output along with the rule. C4.5rules-p can optionally include certainty information in the generated Prolog rules. Standard Prolog does not support the notion of reliability of a statement. To add this information to the Prolog statements in a way that would be understandable to most Prolog systems, we use a pseudo random number generator to cause a rule to fail in proportion to its certainty value. A random integer is generated and tested against the certainty value. A statement can fail if this test fails, no matter what the value of the condition attributes. C4.5 computes the certainty values as a number less than 1 and outputs them with a precision of 0.001. The modified c4.5rules program multiplies this number by 1000 to avoid having to deal with nonintegers. C4.5rules-p outputs the necessary code to handle the certainty value if the user invokes it with a '-p 1' command line argument. We used a different command line argument for this because the user may not always need to have the certainty values in the output. The results are shown in Table 4.

class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15,, A16, 8) :- random(1000, N__), N__ < 793, A10 = 8, A12 = 5, A13 = 3, A14 = 8,
class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15,, A16, 8) :- random(1000, N__), N__ < 793, A7 = 9, A13 = 0, A14 = 9, A16 = 7.
class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15,, A16,

8) :- random(1000, N__), N__ < 707, A6 = 9, A10 = 7, A11 = 6, A13 = 0.

Table 4. Prolog statements with the ceratinty values.

In Table 4, the first rule has a certainty value of 79.3%. The random(1000, N__) function assigns a number between 0 and 999 to N__. This value is then compared to the certainty value of the rule, which is 793. The statement could fail based on the results of the comparison. The random number is named N__ to lessen the chances of an accidental clash with the name of a condition attribute. One could implement the random number generator as follows [5]:

```
seed(13).
random(R, N) :- seed(S), N is (S mod R), retract(seed(S)),
        NewSeed is (125 * S + 1) mod 4096, asserta(seed(NewSeed)), !.
```

We have taken an active approach to representing the certainty values, because they can actually cause the statements to fail. In an alternate implementation, we could choose to simply output these values as part of the statements and leave any specific usage to the user. An example, taken from the last statement in Table 4, would be class(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, N__, 8) :- A6 = 9, A10 = 7, A11 = 6, A13 = 0, N__ = 707.

4 Concluding Remarks

Generating classification rules as Prolog statements allows them to used directly in new contexts. They can be traversed in any direction, and can be readily integrated into Prolog-based systems. With the kind of temporal data explained in the paper, Prolog's searching abilities can be employed to search the rules recursively. The certainty values can be ignored in the output Prolog statements if they are to be used in deterministic environments. The user can also opt to simulate the possible failure of the rules by having the certainty values represented in the rules as thresholds for test against randomly generated numbers.

The modifications in the c4.5rules program retain backward compatibility, and its output is unchanged when the new options are not used. The modifications are available in the form of a patch file that can be applied to standard C4.5 Release 8 source files. It is freely available from http://www.cs.uregina.ca/~karimi or by contacting the authors. C4.5 Release 8's sources are available for download from Ross Quinlan's webpage at http://www.cse.unsw.edu.au/~quinlan/

References

1. Blake, C.L and Merz, C.J., UCI Repository of machine learning databases [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science, 1998.

2. Karimi, K. and Hamilton, H. J., "Learning With C4.5 in a Situation Calculus Domain," *The Twentieth SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence (ES2000)*, Cambridge, UK, December 2000.
3. Karimi, K. and Hamilton, H. J., "Finding Temporal Relations: Causal Bayesian Networks vs. C4.5." *The 12th International Symposium on Methodologies for Intelligent Systems (ISMIS'2000),* Charlotte, NC, USA.
4. Quinlan, J. R., C4.5: Programs for Machine Learning. Morgan Kaufmann, 1993.
5. Clocksin, W.F., Melish, C.S, Programming in Prolog, Springer Verlag, 1984.
6. http://ww.cs.uregina.ca/~karimi/URAL.java