Generalized Fuzzy Rule Induction: An Object-Oriented Approach

Edward S. Blurock
Johannes Kepler University,A-4040 LINZ, Austria
email: blurock@risc.uni-linz.ac.at

June 10, 1999

Abstract

This paper outlines the implementation of generalized fuzzy rule induction methods in an object-oriented framework. A key to the generalization process is the formulation of partial characterizations and probabilities within a generalized logical framework. The logic and the algorithms are implemented in a general way within an object-oriented framework. The pattern of the methods are defined in a super class and specific variations of the algorithms and logical data are either defined in the derived classes or during runtime specifications.

1 Introduction

Data analysis is the extraction of general principles from a history, or 'training set', of known examples, i.e. data instances. Machine learning[11, 12, 2, 8] is one class of data analysis methods and differs from 'classical' methods, such as statistics or pattern recognition, in two ways. First, it is a more 'human' approach in that the methods are based on either biological (such as neural nets or genetic algorithms) or psychological principles (such as the rule induction methods). Secondly, it is usually based on the optimization (iterative, search or genetic) of a class description, as opposed to a deterministic approach as in many multivariate statistical methods.

Rule induction[10] algorithms are a class of machine learning techniques which take a set of instances and a concept to be learned (when the method is 'supervised') as input and produce a rule to describe the concept as output. The language of the 'induced' rules produced is first order predicate logic. The exact form of these expressions depend on the particular algorithm. For example, decision trees are produced by `ID3`[15], `C4.5`[16, 17] and `CART`[1], logical conjunctions are produced by `AQ`[9] (and its derivatives) and version space methods[6], and if-then-else rules are produced by `CN2`[3, 4].

Whatever the representation, the fundamental principle behind rule induction is that when the derived rule is true for the values of a given instance, then the corresponding concept represented by the rule is true. With supervised methods, the search optimizes the derived rule directly with the concept. With unsupervised methods, the training set of instances is 'classified'. Not only are the rules 'learned', but the corresponding 'concepts' represented are discovered; the concept is the set of similar objects and the rule is the corresponding description.

In their original formulation, most rule induction methods are defined using 'crisp' logic, where the membership function, μ, has the output domain $\{0,1\}$, where 0 and 1 represent `false` and `true`, respectively. Many methods, in later formulations, have been generalized to 'fuzzy logic', where the output domain of μ is the interval $[0,1]$. The values 0 and 1 still represent `false` and `true`, but values inbetween represent fuzzy deviations. Values close to 1 are almost `true`, values close to 0 are almost `false` and values around 0.5 are undecided.

This paper describes the philosophy behind translating the crisp logic formulation to a fuzzy logic formulation taking advantage of an object-oriented framework for their implementation. Since the fuzzy logic operations originate from machine learning techniques, viewing the fuzzy operations as mathematical representations of 'linguistic' concepts is not an issue here. The fuzzy logic will have a probabilistic operational semantic.

In this paper, first, the role object-oriented programming techniques play in the generalization process will be outlined. Then, the relationship between partial characterizations and logical objects will be introduced. Then an outline of how the logical representation of partial characterizations is implemented within an object-oriented framework will be made. Finally, the use of the generalized logical structures will be shown in the implementation of the generalized rule induction method of decision trees.

2 Object-Oriented Generalization

An important aspect of generalizing is the recognition of how a group of similar algorithms are related. Through the operational analysis of each individual algorithm and the data structures they entail, common features are recognized. These common features are then used as the basis for the design.

Since the historical development of a class of algorithms is usually an evolutionary process, each algorithm is often only a slight variation of its ancestor. This facilitates the treatment of individual algorithms as special cases of a general algorithm. These differences can take the form of, for example, differing measures (such as the distance function in clustering), steps which may or may not be used (for example, filtering out rules) or additional information tht guides the algorithm.

Through the polymorphism concept of object-oriented design, the implementation of a generalized algorithm, where different specialized algorithms can be derived, is facilitated. Polymorphism is used in algorithm generalization in the following ways:

Hierarchial Structure The algorithms are represented in a hierachial structure where the generalized algorithm is the super class from which the specialized algorithms are derived. The specialized algorithms can also span many levels.

Virtual Operator The generalized algorithm is defined in terms of virtual operators which are then specified or redefined in the derived algorithms.

Member Data The necessary member data used by virtual operators defines the important information pertaining to the structure of the class. When possible, this data is hierarchially structured and the more general data is defined in the super class definition.

Modular Operators Specific algorithm implementations often differ through the substitution of specific external functions. A common example is a measure function. The basic algorithmic structure does not change, but the measure function guides the search.

For example, in the rule induction implementation described here, an important set of data structures describe the logical value data types. The logical value data types are defined in a hierarchial arrangement with a super class `Logical` from which the different logical value data types are derived, such as n-valued logic or different fuzzy logic classes. In defining a rule induction algorithm, only the super class, with its expected virtual functions, of a data type is used. For example, when a logical operation is called for, then the `Logical` class is specified. The necessary logical operations are available, not only for the super class, but also all the derived classes. Thus, the a different implementation is not needed for crisp logic, or any of the fuzzy logic variations. Which modular operator is used defines the behavior of the rule induction method. All these operators are derived from the super class `Operator`. In the definition of the rule induction algorithm, only the super class is specified. Only at runtime is the actual derived operator used.

3 Partial Characterizations and Logic

One purpose of rule induction machine learning methods is to characterize an instance. In a crisp world, an instance either has a characteristic or it does not. For example, an object is red or an object is 'good' and can represented by the predicates, `red(object) good(object)`. If the object has the characteristic, then the corresponding predicate is `true` and if it does not, then the corresponding predicate is `false`.

The characterization of a fixed group of instances, regardless of whether in a crisp or fuzzy[18, 5, 7] formulation, can be accomplished by counting how many of the instances have the particular characteristic. The charcterization of the group is that a certain percentage of these instances have particular characteristics. For example, 80% of the instances of a group are 'red' and 95% of the group are 'good'. If the group of objects is representative of the entire space of all possible instances, which is the fundamental assumption behind learning from examples, then the percentage of objects can be translated to a probability that an object has the characteristic. In the example given, if an object belongs to this group then there would be an 80% chance that the object is red and a 95% chance that the object is good.

The transfer of group characteristics to partial characterizations of single instances can be done by transfering the corresponding probabilities to the individual instances of the group. The original different characterizations of the individual instances within the group are replaced by the characteristics of the group. All instances within the group would have the same partial characterization. In the example above, this would mean that the characterization of an instance within the group is that it has 80% red character and 95% good character. In a sense, the entire group of 'real' instances is represented by a single 'virtual' instance which is the parital characterization; In this case, the set of instances have been generalized to a single instance description.

In the examples given, if an object has a particular characterization and it implicitely assumed that a complementary characterization exists. The object having 80% `red` character is assumed to have 20% `not-red` character. The sum of this characterization and its complement pair is 100%.

A generalization of partial character and its complement, is a partition. A partition, as used here, is a set, not necessarily a pair, of related partial characterizations where their sum is 100%. In crisp logic, the most frequent partition is `true-false`. The `true-false` logic is a special case of `2-valued logic`. The partition for the n-valued logic is a set of predicates, each of which is true when the object has the specified character.

Generalizing this, a partition can be represented by any consistent set of characterizations, where the sum for any given object is one. For example, a partition representing `height`, with `Small`, `Medium` and `Large` can be a set of numeric predicates over the `height` parameter of the object. For example a measured height of 160 centimeters for each of the three predicates would yield .80, .20 and 0 for `Small`, `Medium` and `Large`, respectively.

3.1 Characterization, Logic and Frequencies

The main idea behind the translation of a rule induction method from its classical n-valued formulation to a fuzzy logic formulation, is relating the partial characterization with probabilities and then the relaxation and approximation of the probabilistic concepts to fuzzy logic concepts. Fuzzy logic will continue to hold some 'semantic' of the probability concepts, but some of the axioms associated with probability theory will be relaxed.

Both probability and its logical counterpart have values between 0 and 1. A partition of partial characterizations and a logical partition both have the property that their sum is one. What the logical functions do not have, in contrast to probabilistic functions, are the further axioms and distribution assumptions.

The logical representation can approximate the probabilistic distributions with a simplier function. For example, an exact gaussian function could be approximated by the fuzzy logic triangular ramp function, where the slopes of the ramps are related to the standard deviation.

In crisp logic, the characterization is represented by a predicate, whose value, when applied to the object is either **true**, the object is of that character, or **false**, the object does not have that character. When dealing with a group of objects, to find the probability that a set of objects have a particular character, the number of objects having this character is divided by the total number of objects. The logic value **true** says count the object and **false** says don't count. An equivalent formulation, as a sum, is:

$$p = \frac{\Sigma P(obj)}{(\Sigma P(obj) + \Sigma N(obj))} \tag{1}$$

where, p is the probability of objects having character P, $P(obj)$ is one when the object, obj, has the character P and $N(obj)$ is one when the object does not have the character P.

In fuzzy logic, the partial characterization is also, as in crisp logic, represented as a predicate over the object. However, in this case, the logic value is not discrete, but is a continuous fuzzy value between 0 and 1. Due to the non-discrete values, counting proceeds through the same summation as the crisp logic except that the $P(obj)$ and $N(obj)$ are now fuzzy predicates. For a general partition the probability that a set of objects has the property, p_i, of a partition, P is:

$$p_i = \frac{\Sigma_{obj}\mu_i(obj)}{\Sigma_{partitions\ j}\Sigma_{obj}\ \mu_j(obj)} \tag{2}$$

where μ_i and μ_j are the logic fuzzy logic membership functions representing the partial characterizations, c_i and c_j within a partition, P.

In the classical formulation an object can only be assigned one distinct classification. The conjunction of two classifications reduces down to whether they are the same (true) or they are different (false). In the relaxed form, if one object can be considered a mixture of characteristics, then the conjunction no longer gives a discrete n-Valued result. If the characteristics were of a probabilistic nature, the conjunction of two (evenly distributed) characteristics would be clear:

$$P(A \wedge B) = P(A) \cdot P(B) \tag{3}$$

Thus, since the characterization of the membership functions of the predicates represents a relaxation of probabilistic concepts, the $\wedge$ function would have the same multiplicative form:

$$\mu(A \wedge B) = \mu(A) \cdot \mu(B) \tag{4}$$

Following the same arguements, the disjunction of two objects would be of the form:

$$\mu(A \vee B) = \mu(A) + \mu(B) - \mu(A)\mu(B) \tag{5}$$

3.2 The Fuzzy Logic Semantic

There are several ways in which the predicates and operations can be interpreted within fuzzy logic. Which viewpoint is taken not only influences how the logical are interpreted, but, which is important for this paper, how the logical operations and expressions within a given 'crisp logic' rule induction method can be extended to fuzzy logic. Three standard viewpoints are:

Probability The probability function implies one is dealing with a set of objects. Here, the probability function gives the chances that a certain event occurs. Probability comes into play during 'learning' when counting concepts and characterizations within the training set.

Certainty The certainty function implies that one is dealing with a single object. With what certainty, or accuracy, can I make a statement about the object. Certainty comes into play after a concept has been 'learned' and is being tested or applied.

Membership Function The membership function is used in fuzzy logic. This function represents the logical values with a range from zero to one. This is considered a mathematical function which is the connection between characterization and probability and logic.

Since machine learning is based on analyzing commonalities and differences within a set of examples, the probability is a fundamental principle within these methods. Thus the interpretation of the fuzzy logic descriptions will be closely associated with a probabilistic notion which, in turn, is related to partial characterizations.

4 Object-Oriented Logical Objects

Within a logical framework the partial characterizations are predicate membership functions mapping objects from the parameter space into a logical domain. The logical value objects are arranged in a hierarchical structure with the 'super' class being the class `Logical`. All logical classes have a set of standard virtual operators in common:

AND, OR, NOT These define the basic logical operations associated with the logic of the class. These could be, for example, probabilistically based (`Continuous`), a t-norm (`TNorm`) or maximum, minimum based (`MaxMin`).

getValue(), setValue(v) The actual logic value is represented as a `double` value (usually in the interval $[0, 1]$). These operators access the logical value (and could have checks for its validity).

A set of logical classes and their position in the hierarchy is shown in figure 1. The super class from which all other logical value classes are derived is the `Logical` class. This super class is important, because in the implementation of the rule induction methods only the `Logical` class is specified. The actual instantiation of the logic could be any of the derived logical classes. This is possible because, as shown above, all the derived classes have the same set of virtual operators available.

Some classes have additional information that needs to be specified at runtime before the class is completely defined. For such classes, there are three levels of definition. The first level is the hard-coded program which gives the pattern of information and determines the class's behavior. The second level is where specific information is defined at runtime to fully define the data type. The third level is an actual data value of this new data type.

The `NValuedLogic` class is a simple example of a such a composite class. At the first level, i.e. the class implementation, the information about the possible set of finite logical values is left unspecified. At runtime, the class as a data type is defined by specifying the finite set of values. For example, the 2-valued `TrueFalse` logic data type is defined when the symbols, `True` and `False` are specified. The third level, also specified at runtime, is when a particular logic value, such as `False` is specified (for example, as the result of a predicate).

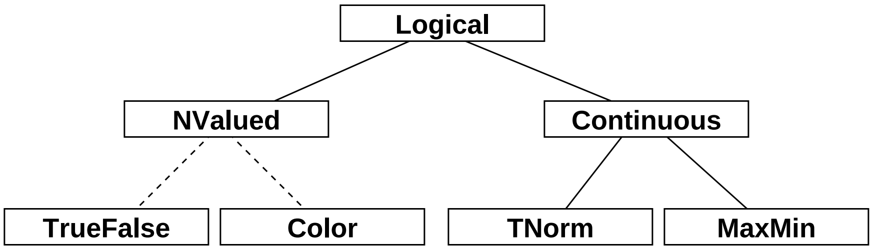

Figure 1: The Hierarchy of logical value objects. The dotted lines from `NValued` denote that these classes were derived dynamically at runtime.

Another important set of data structures is the class of logical predicate functions. These take a parameter value (from an instance or attribute) and return a logical value. A generalized predicate is defined by the name of the instance parameter involved, the functional transformation of the parameter data type to a logical type and the logical type. With the same structure a range of predicates can be defined. Thus, for example, the same data structure can define a crisp logical predicate yielding `true` or `false`, or a fuzzy membership function taking a numeric value and giving a value from zero (`false`) to one (`true`) or even a fuzzy function taking a non-numeric parameter.

As with the `NValued` class, the `Predicate` class has three levels of definition. The `Predicate` class is a composite of the important characteristics of a predicate, namely (see figure 4):

Logical The logical value data type of the result of the predicate. The logic can be any class stemming from the `Logical` class.

Operation This describes the specific transformation from the object in parameter space to the logical value. The function can be any class stemming from `Operation`. Figure 1 shows a hierarchy of operators that could also be used to define a new instantiation of a predicate.

String This is the name of the parameter in the instances specifying the object to be sent to the predicate operation. This member is a simple string name.

The first level is the compiled class code which specifies the pattern of the predicate definition, i.e. the `Logical` and `Operation` compostion. The second level, specifies the exact `Logical` and `Operation` derived classes of the composition. In the example of figure 4, the operation class specified is `Pyramid` and the logic class specified is `Continuous`. At this point, the predicate class defined is a large class of possible fuzzy predicates that can be specified as a ramp function such as less-than, equal-to and greater-than specified (see figure 2). A predicate of this class is fully specified, in the third level, when the instance parameter name is specified and the set of `Pyramid` parameters are specified.

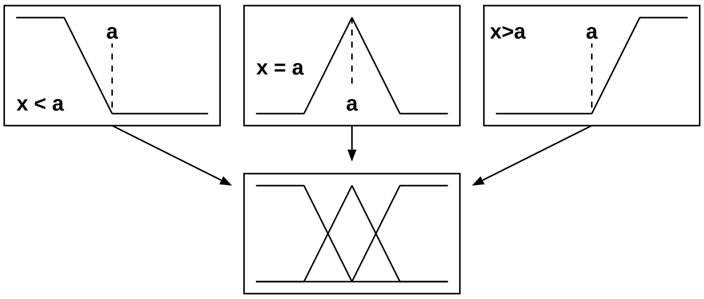

Figure 2: Typical fuzzy (`Pyramid` class) predicate partition. Note that the sum of the values of the partitions is always one.

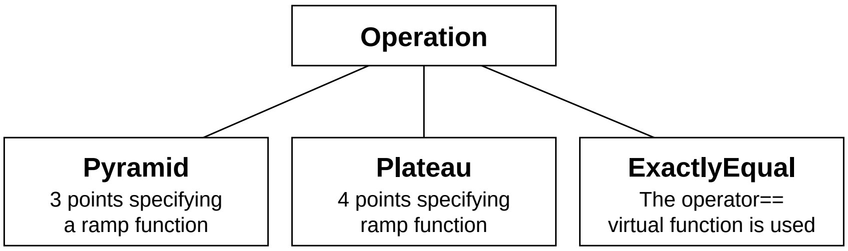

Figure 3: Virtual function classes, derived from `Operator`, that can be dynamically inserted in the `Predicate` class

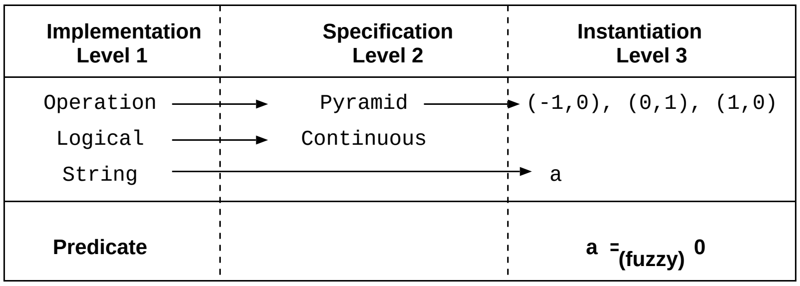

Figure 4: The three levels of definition of a predicate object

5 Example: Generalized Rule Decision Tree

The generalized decision tree method represents a class of supervised rule induction methods derived from the *ID3* method of Quinlan[13, 14, 17, 1]. A decision tree can be thought of as a program of **`if-then-else`** conjunctions. A conjunction, i.e. the condition of a **`if-then-else`**, is true (crisp logic) or to what extent this conjunction is true (fuzzy logic), determines if, or to what extent, the terminal node, the corresponding **`then`** result is true. Said another way, the decision tree in terms of input parameters is equivalent to the goal partition. The heart of the method is the selection of the partitions on each node of the tree. Each partition is used to 'divide' the training set into clusters. Each new cluster represents a new node in the tree. The selection of an expression then continues with these new nodes until a full separation of goal classes is achieved. The result is a rule in the form of a decision tree where the non-terminal nodes are the questions and the terminal nodes represent the decision about the concept.

In a crisp **`true-false`** logic, each non-terminal node is a predicate and the two branches represent the **`true`** and **`false`** answers to this predicate. The nodes of the decision tree 'decide' the answer to the goal predicate. The crisp decision tree has distinct branches, each of which represents a distinct answer to the node predicate. The logic is restricted to n-valued logic, where n is a finite value.

The generalized form replaces the n-valued logic values with a partition. Each branch, instead of corresponding to a logical value in an n-valued logic, corresponds to a predicate in the partition. If the predicate gives a fuzzy logic value (representing a partial characterization), then, instead of an instance answer satisfying only one branch, as in the crisp decision tree, a single instance can satisfy all branches to varying degrees (where the degrees add up to one object, hence the use of a partition). The partial characterizations are viewed as probabilities (as explained earlier). These probabilities, derived from the logical expressions, are used to compute the measure to select the best expression.

The algorithm described by Quinlan selects the 'best' non-terminal node partition on the basis of an entropy measure over a set of training instances. The entropy measure,given a goal partition and a description partition, gives a numeric result from zero to one representing their similarity. The most similar description partition is inserted as the next node in the tree. The process continues with these new nodes. When a node sufficiently describes a single predicate in the goal partition, then the process stops.

5.1 Probability Evaluation

The fundamental operator in the decision tree method decides how close a given partition, E_i, is to the partition, C_i, given a subset of a training set of examples. Essential to deciding this question, using the original ID3 method as a guide, is the evaluation of probabilities and conditional probabilities of the partitions within the training set.

Essential to the definition of the entropy measure are the following probability notions with respect to from the training set of examples:

Function	Probability at node
$P(E_{prev})$	Reaching the node
$P(C_i \mid E_{prev}) = \frac{P(C_1 \wedge E_{prev})}{P(E_{prev})}$	Goal Predicate
$P(E_j \mid E_{prev}) = \frac{P(E_j \wedge E_{prev}}{P(E_{prev})}$	Description Predicate
$P(C_i \wedge E_j \mid E_{prev}) = \frac{P(C_i \wedge E_j \wedge E_{prev})}{P(E_{prev})}$	Goal and Description

Reaching a particular node within the decision tree involves following the nodes and branches from the root node. Following a path through the decision tree is equivalent to evaluating the conjunction of predicate in the path. Thus, if the path from the root node, N_0, to the current node, N_c, is given by $\{N_0, N_1, ...N_{c-1}\}$ and each branch connecting these nodes has a corresponding description predicate, $\{E_0, E_1, ...E_{c-1}\}$, then the logical expresion representing this path is given by:

$$E_{prev} = \wedge_{i=0}^{c-1} E_i \quad (6)$$

This expression is the basis condition for arriving at this node. Thus, it appears in all probabilities and conditional probabilities. Using the conjunction definition of equation 4, to what extent an object reaches the node is given by:

$$\mu(E_{prev}) = \Pi_i \mu(E_i) \quad (7)$$

This expression occured only implicitely in the crisp logic term because the value of E_{prev} is either absolutely `true` or absolutely `false` and $\mu(E_{prev})$ is either zero or one. Thus, only those instances which 'reached' the node, i.e. satisfied the expressions in the previous nodes, were considered. This makes the original method a 'divide and conquer' method because as one moves down the tree, less and less objects are considered. In the general form, however, the sum must proceed over all partial values. Through this generalization, the complexity of the algorithm is increased. The advantage gained by generality is paid for in complexity.

	E_1	...	E_n	Totals
C_1	$P(C_1 \wedge E_1 \vert E_{prev})$	...	$P(C_1 \wedge E_n \vert E_{prev})$	$P(C_1 \vert E_{prev})$
C_2	$P(C_2 \wedge E_1 \vert E_{prev})$	...	$P(C_2 \wedge E_n \vert E_{prev})$	$P(C_2 \vert E_{prev})$
...	.	.	.	
C_m	$P(C_m \wedge E_1 \vert E_{prev})$	...	$P(C_m \wedge E_n \vert E_{prev})$	$P(C_m \vert E_{prev})$
Total	$P(E_1 \vert E_{prev})$	...	$P(E_n \vert E_{prev})$	1.0

Figure 5: Expression/Class Probability Table

The key to the entropy calculation is the table given in figure 5. A partition is similar if the candidate description partition 'divides' the set of instances in the same way that the goal partition does. The best candidate partition would be one where $E_i = C_i$ or in terms of probabilities, $P(C_i \wedge E_i|E_{prev}) = 1$ and $P(C_i \wedge E_j|E_{prev}) = 0$, where $j \neq i$. In this case, the table of figure 5 would be diagonal. The worse case would be $P(C_i \wedge E_j|E_{prev}) = c$, for all i and j, which would mean there is no relation between the candidate partition and the goal partition. The entropy function reflects this idea in a single number.

The entropy, written in terms of these probabilities is:

$$\Sigma_{goal} p_{goal} (\Sigma_{desc} (p_{goal,descr} ln(p_{goal,descr}))) \tag{8}$$

where

$$p_{goal} = P(C_{goal}|E_{prev})$$
$$p_{goal,descr} = P(C_{goal} \wedge P(E_{descr})|E_{prev})$$

As can be seen, the basis of the entropy term is probability and as explained earlier, in this object-oriented generalization, the probability is related to the partial characterization which is in turn related to the membership function of the logical expression. Thus, the probabilities that a given logical statement is true, $P(L)$, is equivalent to the membership function evaluation of the logical statement, $\mu(L)$:

Probability		Logic
$P(E_{prev})$	$=$	$\Pi_n \mu(E_n)$
$P(C_i \vert E_{prev})$	$=$	$\frac{\mu(C_1)\mu(E_{prev})}{\mu(E_{prev})}$
$P(E_j \vert E_{prev})$	$=$	$\frac{\mu(E_j)\mu(E_{prev})}{\mu(E_{prev})}$
$P(C_i \wedge E_j \vert E_{prev})$	$=$	$\frac{\mu(C_i)\mu(E_j)\mu(E_{prev})}{\mu(E_{prev})}$

5.2 Operational Plug-Ins and Algorithm Definition

Another aspect of the generalized decision tree method is the recognition of the 'plug-in' operators which can be independently substituted to define another variant of the decision tree creation algorithm. Their implementation in the object-oriented framework is through modular and virtual operators. Several distinct algorithms within the entire decision tree creation algorithm have been identified:

PartitionGeneration Given the set of objects in the training set, a set of expressions must be generated to be used as candidates for expressions in the nodes of the decision tree. This process can either be dynamic, i.e. the expressions are generated as needed for each node or static, i.e. a list of expressions are generated once at the begining of the procedure.

ComputePartitionMeasure There are several possibiliites for ranking how similar the descriptor predicates are to the goal predicates. In the previous sections, the generalized form of the entropy was used.

ExpandNode? This determines if the node sufficiently describes one of the goal partition predicates. In addition, pruning techniques are used and are an important part of the accuracy of the final decision tree procedure.

The generalized decision tree rule induction method can be defined using, as can be seen in figure 6, the basic structure of the algorithm is similar to the original definition. In fact, if only crisp `NValued` logical structures are used with the entropy definition given above, the original algorithm is defined. However, even with no change to the algorithm definition (i.e. the plug-in routines remain the same), if fuzzy logic predicates are used for data input, a fuzzy decision tree is produced.

GeneralizedDecisionTree ($TrainingSet$ $GoalPartition$)

Input: $TrainingSet$: The set of instances to learn from
$GoalPartition$: The goal partition that the decision tree should describe

(Initialize) $RootNode \longleftarrow InitializeTree()$
(Start) $OpenNodeList \longleftarrow RootNode$
(Loop) **while**($OpenNodeList$ not empty)
  (NextNode) $CurrentNode \longleftarrow GetNodeFromOpenNodeList()$
  (Expand?) **if**($ExpandNode?(CurrentNode)$)
    (Candidates) $Partitions \longleftarrow PartitionGeneration()$
    (Best Partition) $Best \longleftarrow ComputePartitionMeasure(Partitions, GoalPartition)$
    (Add) $Sons \longleftarrow AddNodesToTree(Best)$
    (Open) $OpenNodeList \longleftarrow Sons$

Figure 6: The top level decision tree algorithm

6 Conclusion

The implementation of generalized rule induction methods within an object-oriented framework was described and illustrated with a generalized decision tree method. A key to the generalization is the transfer of partial characteristics and probability concepts to a general 'fuzzy' logical framework. The object-oriented framework, through a hierarchial structuring of data and algorithm classes provides flexibility and implementation generality.

The concepts described in this paper have been implemented (in `C++`) in the object-oriented machine learning framework, **ANALYSIS++**. Further methods, such as statistical, optimization and clustering are being implemented and expanded. As in the work in this paper, these other classes algorithms have also been examined, their essential characteristics analysed and then implemented with a high degree of generality within the object-oriented framework.

References

[1] J.H. Breiman, R.A. Friedman, Olshen, and C.J. Stone. *Classification and Regression Trees.* Wadsworthand Brooks, Belmont CA, 1990.

[2] Garry Briscoe and Terry Caelli. *A Compendium of Machine Learning.* Ablex Publishing Corporation, Norwood, New Jersey, 1996.

[3] P. Clark and T. Niblett. Induction in noisy domains. In *Progress in Machine Learning.* Sigma Press, 1987.

[4] P. Clark and T. Niblett. The cn2 induction algorithm. *Machine Learning,* 3(4):261–284, 1989.

[5] D. Driankov, H. Hellendoorn, and M. Reinfrank. *An Introduction to Fuzzy Control.* Springer-Verlag, 1993.

[6] H Hirsch. Generalizing version spaces. *Machine Learning,* 17(1):5–46, 1994.

[7] R. Kruse, J. Gebhardt, and F. Klawonn. *Foundations of Fuzzy Systems.* John Wiley and Sons, 1994.

[8] Pat Langley. *Elements of Machine Learning.* Morgan Kaufmann Publishers, Inc., 1996.

[9] R.S. Michalski. Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 2(4):349–361, 1980.

[10] Ryszard S. Michalski and Robert E. Stepp. *Machine Learning: An Artificial Intelligence Approach,* chapter Chapter 11: Learning From Observation: Conceptual Clustering. Morgan Kaufmann Publ. Inc., 1983.

[11] Ryzard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach.* Morgean Kaufmann Publishers, Inc., 1983.

[12] Tom M. Mitchell. *Machine Learning.* WCB/McGraw-Hill, 1997.

[13] J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.

[14] J.R. Quinlan. Decision Trees and Decision Making. *IEEE trans. on Systems, Man and Cybernetics*, 20:339–346, 1990.

[15] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):293–266, 1990.

[16] J.R. Quinlan. Determinate literals in inductive logic programming. In *Proceeding of the Twelfth IJCAI*, pages 746–750. Morgan Kaufmann, 1991.

[17] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1992.

[18] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.