Decision Tree Learning for Drools

by

Gizil Oguz

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

École Polytechnique Fédérale de Lausanne

August 2008

Author .....
Department of Computer Science
August 2008

Supervised by .....
Mark Proctor
JBoss Rules, Drools
Thesis Supervisor

Supervised and Certified by .....
Viktor Kuncak
EPFL, School of Computer and Communication Sciences
Thesis Supervisor

To My Bocuk

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Knowledge representation is the area of **artificial intelligence** (AI) concerned with how knowledge is represented and manipulated. Expert Systems use Knowledge representation to facilitate the transformation of knowledge into a knowledge base which can be used for reasoning - i.e. we can process data with this knowledge base to infer conclusions. Today the expert systems are used in many areas such as business, science, engineering. JBoss Rules, Drools is one of these systems. Drools is a Rule Engine that uses the Rule Based approach to implement an Expert System [17]. Drools can be more correctly classified as a Production Rule System. The 'brain' of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data, against Production Rules, also called Productions or just Rules, to infer conclusions which result in actions.

The users can construct the rules by themselves. Normally the experts do that. However, in the case that there is huge number of data available or the list of available data is constantly changing it becomes almost impossible for human-beings to come up with rules that explain the system. There are expert systems allow the system to learn rules. The system builds rules from data. Our aim is to create a system which can learn the rules by analyzing the data. The system should be capable of extracting the patterns from the data. We design the system to give hints about the data to the users which use the Drools Rule Engine.

Decision trees are useful for classifying objects because of their hierarchical nature between the parent and children nodes. A decision tree is both a knowledge representation scheme and a method of reasoning about its knowledge. In this thesis we will construct the decision trees using a learning algorithm. This hierarchical nature between the parent and children nodes of a decision tree will construct the rules that we need for our expert system.

The difficulty of the problem comes from its nature. Huge amount of data is hard process. Even if the number of data is not big it is very likely that the data is very complex, i.e, it has many dimensions i.e. the number of attributes. The data from of which we are trying to extract the information never covers the entire domain so it is not complete. The number of possible rules that we get increases with the number and the complexity of data. Thus, our search space is gigantic and we are trying to discover 'good' rules which are short, meaningful and human-readable. This thesis makes the following contributions:

- We unite the classic machine learning approach Decision Trees with the Rule Based Expert System
- We show that there might be many trees which have the same classification results but different sizes. The Drools needs the decision trees to decrease the complexity of its rule base. The decision trees constructed using the Decision Learning algorithm tries to optimize

the size of the tree by pruning. Moreover, there is a heuristic function which the algorithm adopts to select the best attribute at every split. The heuristic function and the pruning process aim to eliminate repeating nodes during learning in order to create smaller trees. We show that the smaller tree improves the performance of Drools since we transform it to a smaller RETE Network.

- We show that it is possible to create structured rules for the RETE network using structured data. Structured rules decrease the number of intra-element conditions, i.e., AlphaNodes, while the number of inter-element conditions, i.e. JoinNodes, increases due the structured nature.

Here we give an outline of what will be in each chapter of this thesis. The second chapter is background on the problem and the basic terminology of the Drools. We explain how the rules produced by a machine can be useful for the Drools.

The third chapter describes our GO1.0 Decision Tree Algorithm. It first defines the relational machine learning terminology used for the algorithms. Then it gives the details on the different modules.

The fourth chapter gives the implementation details of the GO1.0 Decision Tree Algorithm. We give an overview of the system and describe each module. We explain the main functionality of each modules and their interactions.

The fifth chapter focuses on the experimental results. First we explain each database we use. Second we visualize the rules generated from the decision trees and the RETE network as a result of the integration. We show the differences between the rules generated from structured and simple data and give the classifications results. Then we present the comparisons between different approaches that we explain in the third chapter. We analyze the results of the learners. We compare the classification results of different builders. We present the affect of the pruning algorithms on the results.

The sixth chapter describes related work.

The seventh and last chapter is conclusions an the possible future work.

Chapter 2

Background

2.0.1 Machine Learning

As a broad subfield of artificial intelligence, machine learning is concerned with the design and development of algorithms and techniques that allow computers to approximate functions given by data sets. At a general level, there are two types of learning: inductive, and deductive. Inductive machine learning methods extract rules and patterns out of massive data sets.

The major focus of machine learning research is to extract information from data automatically, by computational and statistical methods. Hence, machine learning is closely related not only to data mining and statistics, but also theoretical computer science. There are different type of learning algorithms which are supervised, unsupervised and reinforcement.

2.0.2 Decision Trees and Decision Tree Learning

Decision trees were first generated manually to make optimal decisions. As a field of AI we use the learning algorithms to generate the models from the data without the help of human or with minimum help. The learning algorithms can also be used to generate decision trees and then they are called 'Decision Tree Learning' algorithms. Using a learning algorithm necessitates a heuristic in order to evaluate the available opportunities and to decide on the one closest to the optimum solution. However, this never guarantees the optimal solution, the optimal model.

2.0.3 Drools: JBoss Rules

Rule Engine

As we pointed before, Drools is a Rule Engine that uses the Rule Based approach to implement an Expert System [17]. A better classification for Drools is a Production Rule System. Production Rules System has an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data, against Production Rules, also called Productions or just Rules, to infer conclusions which result in actions. Inference Engine matches the new or existing facts against Production Rules, that process is called Pattern Matching. There are a number of algorithms used for Pattern Matching by Inference Engines. Drools implements and extends the RETE algorithm. The Drools Rete implementation is called ReteOO, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for Object Oriented systems.

```
rule "name-of-this-production"
  when
     LHS    #One or more conditions
  then
     RHS    #One or more actions
end
```

Figure 2-1: Rule Definition: Production

Rule . It is composed of two parts, i.e. left-hand-side **LHS** is a set of productions which contains the unordered sequence of the patterns (= conditions) and right-hand-side **RHS** (= actions) as given in the Figure 2-1. It uses First Order Logic for knowledge representation.

The RETE Match Algorithm

Drools implements the RETE Algorithm [8]. The word RETE is latin for 'net' meaning network. The RETE algorithm can be broken into 2 parts: rule compilation and runtime execution. The compilation algorithm describes how the Rules in the Production Memory generate an efficient **discrimination network**, i.e., RETE Network. A discrimination network is used to filter data. The idea is to filter data as it propagates through the network. At the top of the network the nodes would have many matches and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. [8] describes 4 basic nodes: root, 1-input, 2-input and terminal.

The rules of a production rule system forms the Rule Base. The data operated on by the rules is held in a global data base called Working Memory. The interpreter executes a production system by performing the following operations.

1. *Match.* Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory.

2. *Conflict resolution.* Select one production with a satisfied LHS; if no productions have satisfied LHSs, halt the interpreter.

3. *Act.* Perform the actions in the RHS of the selected production.

4. *Goto 1.*

[6] defines the RETE network basically consisting of two parts, i.e. Alpha and Beta networks. AlphaNodes having 1 input define intra-elements conditions and form the Alpha network. BetaNodes having 2 inputs define inter-element conditions and form the Beta network (Figure 2-2). Rete performs joins with BetaNodes to calculate cross products.

The Rete network starts with the root node called **ReteNode**, which all objects must enter. It then splits a branch a node per object type; i.e. the first level of discrimination is object type (Figure 2-3).

After the **ObjectType** discrimination node we have one or more Alpha discrimination nodes, each alpha node applies a literal constraint (Figure 2-4).

Then finally **BetaNode** determines the possible cross product for a rule of Cat and Person in the Figure 2-6.

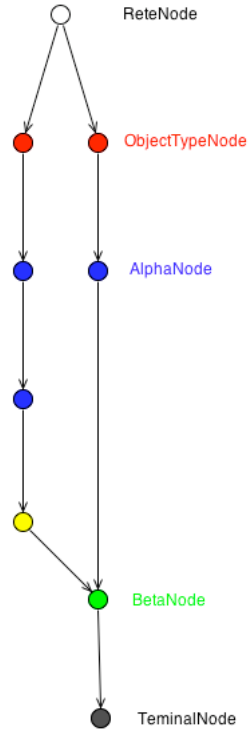

Figure 2-2: Rete Network and Nodes

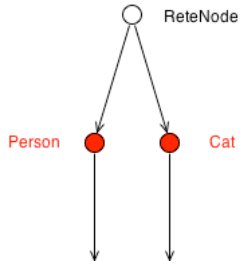

Figure 2-3: ObjectTypeNodes

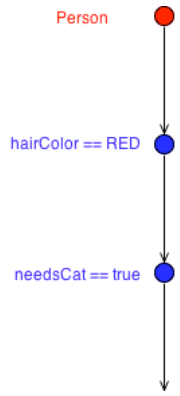

Figure 2-4: AlphaNodes

```
rule "Red-haired Person who needs a Blue-fured Cat without an owner at the same location "
    when
        person: Person(pLocation:location,  hairColor == Person.RED,
                                            needsCat== true, age >15)
        cat: Cat(cLocation:location, furColor == Cat.RED,
                                    needsOwner == true, location == pLocation )
    then
        #actions
        System.out.println("dying your hair to " + "blue" );
        cat.setFurColor(Cat.BLUE);
end
```

Figure 2-5: Rule example for a JoinNode

BetaNode is also called **join node** and it has a left and right memory. All inserted Cats who need an owner and whose fur color is red are remembered by the left memory then as a Person who needs a cat and whose hair color is red is inserted into the system it iterates over that left memory joining with each Cat in turn and propagating to the terminal node. So the terminal node will execute for each matching Cat and Person cross product. This is just like saying:

Program 1 A query of 'Red-haired Person who needs a Blue-fured Cat without an owner at the same location'

```
SELECT * from Person p, Cat c
WHERE p.location = c.location & p.needsCat == true &
      p.hairColor == Person.RED & c.furColor == Cat.RED & c.needsOwner==true
```

The drl is given in the Figure 2-5 and the rete network in the Figure 2-6.

Any objects propagate through the network and pass all these constraints then reaches the terminal node and the rule fires. Terminal nodes are used to indicate a single rule has matched all its conditions - at this point we say the rule has a full match.

Node Sharing Drools also performs node sharing. Many rules repeat the same patterns, node sharing allows the Rete algorithm to collapse those patterns so that they don't have to be re-evaluated for every single instance. Node sharing avoids duplicate value tests. As a result there is less memory usage and the matching algorithm executes faster. For example, if you have two rules all of which have hairColor == Person.RED there will only be one node and that one node would be shared by both rules. That one shared node would then have two outputs to the rest of the nodes that match each of the two rules. The two rules given in the Figure 2-7 share the first two same patterns, but not the last.

The compiled Rete network shows the alpha node is shared as you can see in the Figure 2-8. Had the last pattern been the same it would have also been shared. On the other hand, the beta nodes are not shared like the alpha nodes as you can see in the Rete network (Figure 2-9) which is created from the rules given in the Figure 2-10. Each beta node has its own TerminalNode.

However, when the orders of the conditions change Rete Network is not capable of node sharing any more. Even if the two rules given in the Figure 2-11 share the two patterns, i.e. name== 'Mark' and hairColor == Person.RED, since the pattern of hairColor == Person.RED is not in the same order in both rules there are two different nodes for these conditions and only the first shared node has two outputs to the rest of the nodes that match each of the two rules as you can see Rete network in the Figure 2-12.

As a result, there is no node sharing left if none of the same patterns are in the same order as given

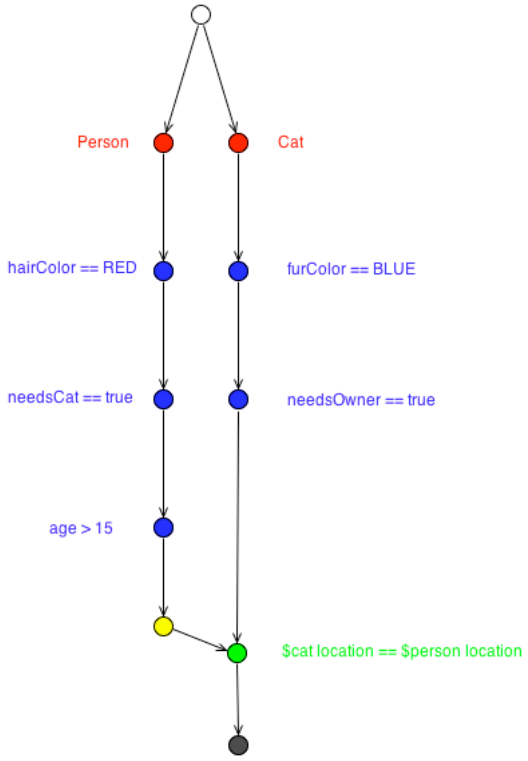

Figure 2-6: The Rete Network of the Example in the Figure 2-5

```
rule "Credit denied "
  when
     Person($pName: name, name== "Mark", hairColor == Person.RED, age <18 )
  then
     System.out.println("credit denied for " + $pName);
end

rule "Dying hair to blue "
  when
    $person: Person(name== "Mark", hairColor == Person.RED, age >15)
  then
    System.out.println("dying your hair to " + "blue" );
    $person.setHairColor(Person.BLUE);
end
```

Figure 2-7: The DRL file of Node Sharing Example

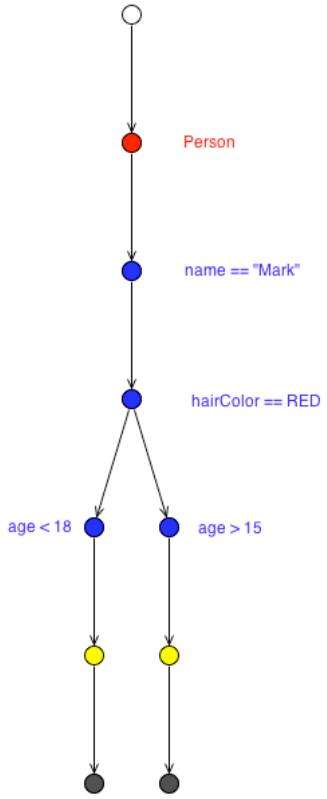

Figure 2-8: The Rete Network of Node Sharing, Figure 2-7

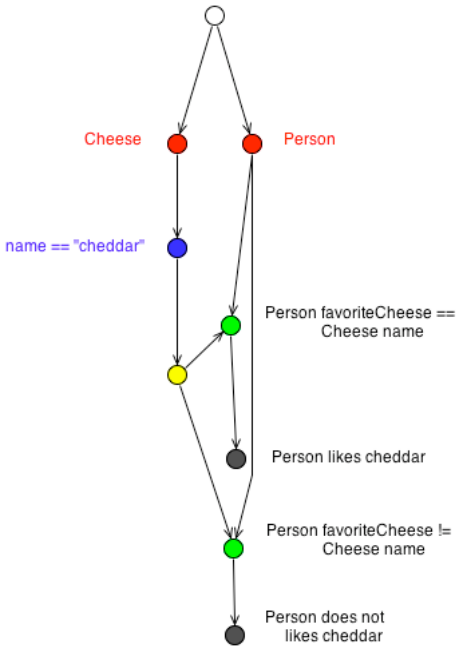

Figure 2-9: Example: Join Nodes are not shared

```
rule "Favorite cheese "
  when
     Cheese( $chedddar : name == "cheddar" )
     $person : Person(favouriteCheese == $chedddar)
  then
     System.out.println( $person.getName() + " likes cheddar" );
end

rule "Not favorite cheese "
  when
    Cheese( $chedddar : name == "cheddar" )
    $person : Person(favouriteCheese != $chedddar)
  then
    System.out.println( $person.getName() + " does not like cheddar" );
end
```

Figure 2-10: The DRL File of Figure 2-9, Join Nodes are not shared

```
rule "Credit denied "
    when
        Person($personName: name, name == "Mark", hairColor == Person.RED, age <18 )
    then
        #actions
        System.out.println("credit denied for " + $personName );
end


rule "Dying hair to blue "
    when
        $person: Person(name == "Mark", age >15, hairColor == Person.RED)
    then
        #actions
        System.out.println("dying your hair to " + "blue" );
        $person.setHairColor(Person.BLUE);
end
```

Figure 2-11: The DRL File for Not Node Sharing

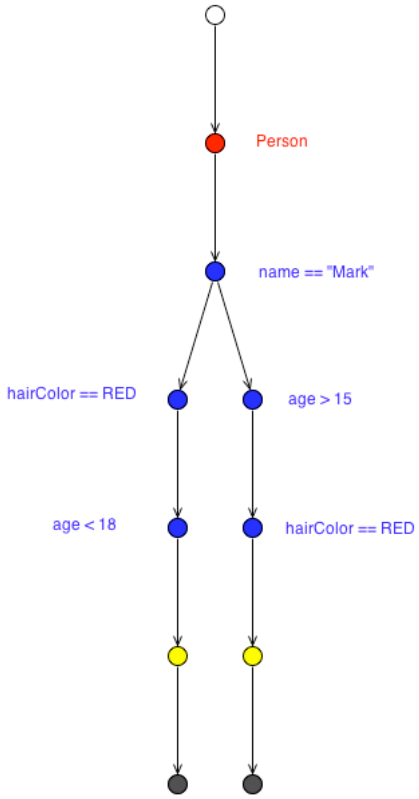

Figure 2-12: The Rete Network for Not Node Sharing Figure 2-11

in the Figure 2-13. Thus, none of the nodes are shared and there are three different node for all the conditions even if the two are the same pattern as you can see Rete network in the Figure 2-14.

```
rule "Credit denied "
    when
        Person($pName: name, age <18, name== "Mark", hairColor == Person.RED)
    then
        System.out.println("credit denied for " + $pName);
end

rule "Dying hair to blue "
    when
        $person: Person(age >15, name== "Mark", hairColor == Person.RED)
    then
        System.out.println("dying your hair to " + "blue" );
        $person.setHairColor(Person.BLUE);
end
```

Figure 2-13: The DRL File of Node Sharing: Worst case

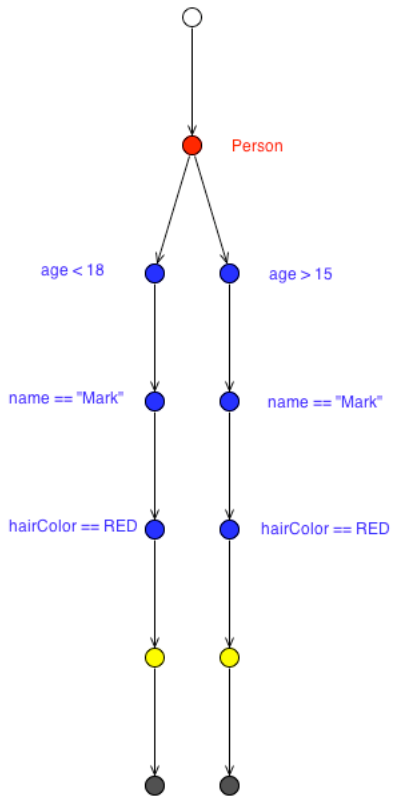

Figure 2-14: The Rete Network of Node Sharing: Worst case Figure 2-13

Chapter 3

The GO1.0 Decision Tree Algorithm

3.1 Relational Vocabulary

The terminology has been constructed based on the glossary of [21]. A *data set* consists of a schema and a set of instances matching the schema. An *instance* is a single object of the world from which a model will be learned. These instances are described by feature vectors and there exist relations between instances or between parts of instances. A *schema* is a description of a data set's attributes and their properties. An *attribute* is a quantity describing a feature of an instance. Each attribute has a domain defined by the attribute type, which denotes the values that can be taken by an attribute. An attribute domain keeps the information of the attribute name and the list of possible values that the attribute can be assigned to.

A *feature* is the specification of an attribute and its value. For example, color is an attribute. 'Color is blue' is a feature of an instance. Many transformations to the attribute set leave the feature set unchanged (for example, regrouping attribute values or transforming multi-valued attributes to binary attributes).

The most common domain types are categorical and quantitative.

1. *Categorical (discrete) domain* commonly constructed by a set of nominal values. This has to be a finite numbered set with values which are discrete. The type nominal denotes that there is no ordering between the values, such as last names and colors. For example, an attribute with the values low, medium, or high. We assume that the attributes have categorical domains by default. But if the user specifies then we can process the domain as quantitative (Section 4.3.1).

2. *Quantitative (continuous) domain*: Commonly, subset of real numbers, where there is a measurable difference between the possible values. Integers are usually treated as continuous in practical problems. This type of domain has to be discretized by defining a various number of thresholds (intervals) for each possible class. Numerical attributes which are a set of real numbers can have a categorical or quantitative domain. In the case of a continuous domain it has to be discretize/categorized by defining a various number of thresholds (intervals) for each possible class.

   For example: $age < 15 \Rightarrow child$
   $15 \leq age < 20 \Rightarrow teenage$
   $20 \leq age \Rightarrow adult$

Literal attributes which are set of Strings can have a categorical or continuous domain. In the case of a continuous domain it has to be categorized by defining a various number of sets for each possible class.

For example: $letter \in \{a, e, i, o, u\} \Rightarrow vowel$
$letter \notin \{a, e, i, o, u\} \Rightarrow consonant$

3. *Complex domain* implements a domain of an attribute that belongs to another object class. We process the complex domains by calling their own classes and their parents. We process all the complex domains by getting their attributes until every complex attributes is converted to set of attributes with categorical or quantitative domains. The Structured schema takes care of this conversion explained in the Section 4.1.1.

3.2 Multi-Relational Decision Tree Learning

3.2.1 The Target Instance, Its Attributes and the Target Attribute

The data structure in Multi-Relational Learning can consist of several object classes which describe particular objects' attributes. However, there is still one object class which is the focus of the classification. The user must choose the object type to analyze by selecting one of the object classes as the target class. Each object with the chosen class will be transformed into a single instance in the data set. After choosing the target class the user should assign an attribute on the class in order to define the labels of the instances. As Quinlan points out one must predefine the target attribute and its categories to which the algorithm will assign the instances for the C4.5 algorithm [18]. This is called supervised learning.

This attribute which is called the target attribute can be a direct attribute of the target class, an attribute of one of the target class attributes or can be constructed by the user as a function on the target class (see Section 4.3.1)
First, given the object class of the target class, we construct a schema defining the structure of the class. Then we process the objects belonging to that class in order to find most relevant and simple patterns, which can explain or predict the label of a future object belonging to the same class. The patterns contain attribute-value descriptions and the structural relation between the classes and their complex domains. These patterns are also called multi-relational or structured patterns.

3.2.2 Training Decision Trees

ID3 Learning Algorithm

Quinlan designed the ID3 (Iterative Dichotomiser 3) algorithm to generate decision trees by processing the learning data [22]. The algorithm intends to produce smaller decision trees. However, it does not always produce the smallest tree since it uses a heuristic as a splitting criterion. The splitting criterion heuristic is based on the information entropy.

C4.5 Learning Algorithm

Quinlan presents the C4.5 Learning algorithm for decision trees in [18]. Compared to the ID3 the C4.5 learning algorithm can tackle with harder domains that contain many number of possible values. We describe the domains in the Section 3.1. C4.5 deals with the numeric (integer or real) and continuous attributes using a discretization technic based on entropy. The discretization algorithm of the continuous attributes is described in the Section 3.2.2.

We use the algorithm called **Learner** to train a decision tree as explained in the Algorithm 3.2.2.

LEARNER($\mathbf{X}, \mathbf{t}, \mathbf{d}$)

```
1  Input:
             X : a set of N labeled instances, and their distribution on target
             t : tree specification, i.e.,
                      attrs: list of attributes,
                      attr_target: target attribute, and
                      C[attr_target]: list of target attribute categories
             depth: depth of branch
2  Select winner from C[attr_target]
             winner ← majority of votes of instances X on C[attr_target]
3  if vote(X) = winner, ∀X ∈ X
4     then return leaf_classified(winner)
5  if attrs = ∅
6     then return leaf_noAttributeLeft(winner)
7  Choose attr_best to split X using heuristic function
8  if ∃ criteria c ∈ list of Stopping Criterion : c = true
9     then return leaf_majority(winner)
10 Create node_New and Split X by C[attr_best]
             X^c ← instances at category c
11 for c ∈ C[attr_best]
        do
12           Instantiate tree specification t' excluding attribute attr_best
             attrs ← attrs − attr_best
13           if X^c ≠ ∅
14              then node_child ← LEARNER(X^c, t', depth +1)
15                   Set node_child as child of node_new at c
16              else Create empty leaf_empty(winner)
17                   Set leaf_winner as child of node_new at c
18 return node_new
```

Figure 3-1: Algorithm: Learner

The learner chooses the best attribute which differentiates the target attribute categories the most. When the learner chooses the best attribute $attr_{best}$ it divides the instances into subgroups so as to reflect the attribute categories of the chosen node at the line 7. Then it creates a separate tree branch for each category of the chosen attribute. For each subgroup the learner calls herself if there is no termination condition satisfied as in the line 7, and the lines 8 and 13. The learning algorithm can terminate by returning a node classifying winner target category if

- all instances vote on same the category in the line 3,

- there is no attribute left to classify in the line 5,
- there is no instance left to classify in the line 16,
- there exists a stopping criteria equal to true given the current statistics, i.e., depth d, the result of heuristic, or number of instances matching the node in the line 8 and the line 13.

The Learner trains herself on the training data. Then the builder tests the result decision tree on a test data set. If test data is not available, it can use a cross-validation on the training data.

Splitting Criteria

There are many heuristic functions as disparity/impurity measures i.e. information gain, gain ratio, gini coefficient, or chi-squared test that we can use as a splitting criteria . During the learning we use the heuristic function in the line 7 in the Algorithm 3.2.2. In the C4.5 algorithm Quinlan adopts the information-theoretic approach which is the information gain (entropy). There is also information gain ratio which is introduced as less biased since it is normalized by the attribute's self information . Breiman et al. initially tries an information-theoretic criterion, but chooses 'Gini' index.
However, Breiman et al. remarks that '. . . within a wide range of splitting criteria the properties of the final tree selected are surprisingly insensitive to the choice of splitting rule. The criterion used to prune or recombine upward is much more important.' [4].

Continuous Attribute Discretization

There are mainly two approaches of discretizing the continuous attributes. One approach is using a global discretization algorithm, which results in a smaller decision tree. However, a global discretization algorithm would ignore the relation of the continuous attribute with the other attributes. The other approach is at any node of tree dicretizing the continuous attribute on the current set of instances that means applying the global discretization algorithm during the training of the decision tree.

The splitting criteria mentioned in the Section 3.2.2 can be utilized in discretization. These splitting criteria are disparity/impurity measures, i.e., information gain, gain ratio, gini coefficient, or chi-squared test. Quinlan points the weakness of C4.5 in domains with continuous attributes and as a solution he presents the Minimum descriptive length, **MDL**, metrics [20]. We adopt the MDL method with information gain presented in [7]. Here we describe a global discretization approach using the MDL method:

1. *Sort the instances on the attribute of interest*
2. *Look for potential cut-points.* Cut points are points in the sorted list where the class labels and the attributes values change.
3. *Evaluate the effectiveness of the discretization.* Calculate the performance of the discretization instance using the disparity measure on each of the cut points, and choose the one with the best performance.
4. *Repeat recursively in both subsets (the ones less than and greater than the cut point value) until*
   (a) either the subset is pure i.e. only contains instances of a single class,
   (b) or some stopping criterion is reached (maximum number of branching is reached)

3.2.3 Building Decision Trees

Given a set of N instances, each belonging to one of K classes, $(x_1, y_1), \ldots, (x_N, y_N)$ where $x_i \in X, y_i \in Y$ such that $Y = \{1, 2, \ldots, K\}$ there are different ways to populate the decision trees. The simplest approach is building a single tree and testing it. The other two approaches are **bagging** and **boosting** which builds multiple trees. Multiple Tree systems, forests, use a voting system to decide the classification of an instance. We partition the set of N instances into learning set L which consists of less instances and a testing set.

Decision Tree Forests: Bagging

We will form a decision tree, $D(x, L)$, using our training procedure on this learning set — if the input is x we predict y by $D(x, L)$. This is called boostraping.If y is a class label, let the $\{D(\mathbf{x}, L)\}$ vote to form $D_B(\mathbf{x})$. Breiman calls this procedure "**b**ootstrap **agg**regating" and use the acronym **bagging** [3].
Assuming we use the full data set consisting of N observations for training we explain the algorithm used to construct a decision tree forest in the Algorithm 3.2.3. The $\{\mathbf{L}^{(B)}\}$ form replicate data

BAGGING($\mathbf{X}, \mathbf{T}$)

Input:
$\mathbf{X}$: a set of N labeled instances
attrs: list of attributes
T : number of trees to grow

1 **for** $t \leftarrow 1$ **to** T **do** — for each decision tree
2 Draw a random sample, $\mathbf{X^t}$, of size N from the learning set, $\mathbf{X}$, with replacement — Boostraping
3 Fit a classifier $D^{(t)}(\mathbf{X})$ to the random sample, $\mathbf{X^t}$
$D^{(t)}(\mathbf{X}) \leftarrow \text{LEARNER}(\mathbf{X^t}, \mathbf{attrs}, \mathbf{0})$.
4 **return** $C(\mathbf{X}) = \arg\max_k \sum_{t=1}^{T} 1_{\{D^{(t)}(\mathbf{X})=k\}}$ — Aggregation by voting

Figure 3-2: Algorithm: Bagging

sets, each consisting of N instances, are drawn at random, but with replacement, from L. Each $(\mathbf{x}_i, y_i)$ may appear repeated times or not at all in any particular $\{L^{(B)}\}$. Some instances will be selected more than once, and others will not be selected. On average, we expect to select about 2/3 of the instances by the sampling. The remaining 1/3 of the instances are called the "out of bag (OOB)" instances. Deterministic learning algorithms tend to overfit. Bagging tries to avoid this by randomizing the input of the learning algorithm.

Boosting Decision Trees

Boosting is a very successful technic for solving the binary classification problems. It was first introduced by Freund and Schapire (1997) as the AdaBoost algorithm [9].

Given a set of N instances, each belonging to one of K classes, $(x_1, y_1), \ldots, (x_N, y_N)$ where $x_i \in X, y_i \in Y$ such that $Y = \{-1, +1\}$ here is the AdaBoost algorithm:

AdaBoost($\mathbf{X}, T$)

Input:

$\mathbf{X}$: a set of N labeled instances

attrs: list of attributes

T : number of trees to grow

1 Initialize the weights

$w_i \leftarrow 1/|\mathbf{x}|$, $i = 1, 2, \ldots, |\mathbf{x}|$.

2 **for** $t \leftarrow 1$ **to** T

do

3 Fit a classifier $D^{(t)}(\mathbf{X}) : X \rightarrow \{-1, +1\}$ to the training data, $\mathbf{x}$, using weights $\mathbf{w}$.

4 Compute the error of the classifier

$$err^{(t)} = \sum_{i=1}^{N} w_i \cdot 1_{\{D^{(t)}(x_i) \neq y_i\}} / \sum_{i=1}^{N} w_i$$

5 Compute

$$\alpha^{(t)} = \log \frac{1 - err^{(t)}}{err^{(t)}}$$

6 Update the weights

$$w_i \leftarrow w_i \cdot \exp\left(\alpha^{(t)} \cdot 1_{\{D^{(t)}(x_i) \neq y_i\}}\right),\ i = 1, 2, \ldots, N.$$

7 Re-normalize the weights, $\mathbf{w}$

8 **return** $C(\mathbf{x}) = \arg\max_k \sum_{t=1}^{T} \alpha^{(t)} \cdot 1_{\{D^{(t)}(\mathbf{X}) = k\}}$ ▷ The final hypothesis

Figure 3-3: Algorithm: AdaBoost

3.2.4 Re-Training Decision Trees

We need a machine that can continuously learn over time. Thus, the current set of rules can change as a result of new objects fed to the working memory. Whenever more objects are fed to the working memory the decision tree learning algorithm should continue from the last decision tree found. A simple and dump way is training a decision tree from scratch by keeping whole instances in the memory. The faster way of re-training decision trees is saving the trees with its matching instances to the nodes. However, this cause memory issues when the number of instances increases. The decision tree learning algorithm is constructive, by its nature it is not iterative. Thus, even if we save the nodes with the list of matching instances whenever the split attribute changes we need to construct that branch from scratch. That means whole tree can be reconstructed if the heuristic chooses a different attribute at the root of the tree. As a result, if some branches of the tree change the rules can completely change. This can remove some rules or add some rules.

3.3 Over-fitting and Right Sized Trees

In supervised machine learning the learning algorithm trains the decision tree using a set of instances, i.e. exemplary situations for which the desired output is known. The set of instances are called *learning data set*. Although the decision tree experiences only the instances in *learning data set* it is expected to be able to predict the correct output for the future examples. Thus, it should be able to generalize to situations not presented during training (based on its inductive bias). However, it is very likely that the decision tree T is perfectly consistent with the learning

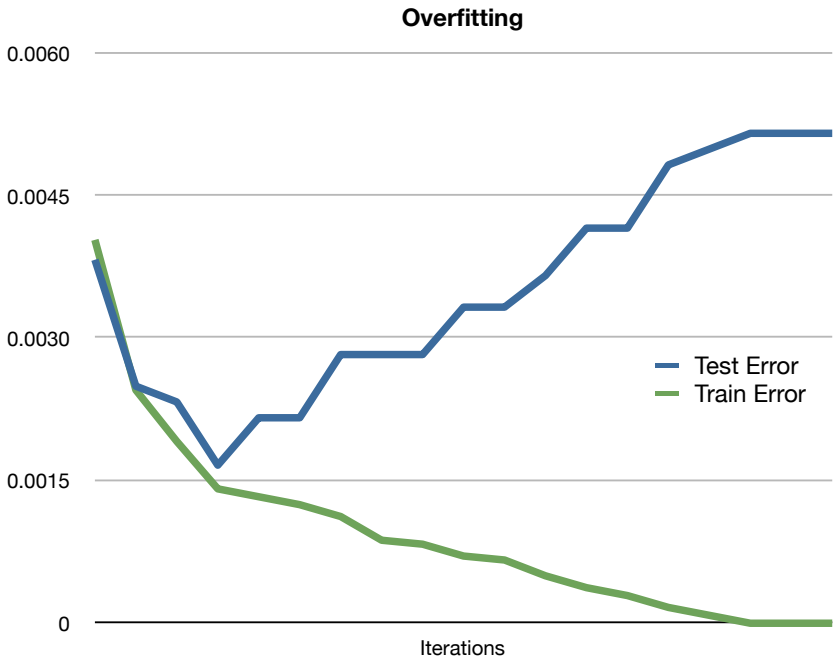

Figure 3-4: The Overfitting

data set L and it will not be good enough to predict classes of unseen data (*test data set*) from the same distribution unless some special care is given to the problem. A decision tree T over-fits the *learning data set if and only if* $\exists T'$ such that:

$$err(L, T) < err(L, T')$$
$$err(L', T) > err(L', T')$$

where L' is the unseen data from the same distribution, i.e. *test data set*.

In machine learning, the problem is called **over-fitting**. Over-fitting is building a statistical model that has too many parameters. As a result, the classifier over-fits the *learning data set* when it adjusts to very specific features of the *learning data set* without generalization. In this process of over-fitting, the performance on the training examples still increases while the performance on unseen data becomes worse. Thus, the over-fitting should be avoided by some extra cautions during the training process. We visualize the training and test errors during the training of a classifier in the Figure 3-4. The classifier starts over-fitting the *learning data set* when the test error starts increasing since the Learner continues training the classifier on the *learning data set* and the training error decreases. Before the point where the test error is minimum the classifier under-fits the *learning data set* since it is not complex enough.

Reasons for Over-fitting. There are many reasons for over-fitting. Especially in cases where learning was performed too long or where training examples are rare the over-fitting is unavoidable. Some other reasons can be

- Noisy data
- Incomplete data (not all cases are covered)
- Given attributes are not enough to predict the outcome

- Not enough data for some part of the learning sample

3.3.1 Pruning Algorithms

The most common way of avoiding over-fitting in decision trees is pruning the decision tree. There are two approaches for the pruning.

- *Pre-pruning* stops growing the tree during the learning, before it reaches the point where it perfectly classifies the learning data set.

  The generic decision tree learning algorithm learner continues splitting the nodes as long as there is an attribute to split and/or the data is not classified perfectly. Adding to the general terminal conditions a node will not be split with the pruning if

  - The number of instances matching to the node is too small (N_t)
  - The impurity of the split on the node (I_t) is low enough
  - The best test is not statistically significant (according to some statistical test)

  The main concern about the pre-pruning approach is that the optimum values of the parameters (N_t, I_t, significance level) are not only problem dependent but they can also differ for the different branches of the same tree.

- *Post-pruning* allows the tree to over-fit and then prunes the tree later. This approach needs a test test to be able to evaluate the generalization error. Unless there is test set available the learning data set L is split into two sets: training set L_T to build the tree and validation set L_V. After building a complete tree T_0 from the training set L_T a sequence of trees $\{T_1, T_2, \ldots\}$ are computed by removing some subsets of nodes from the initial tree T_0. In the sequence each tree T_i is obtained by removing some subtree from the previous tree T_{i-1}. At the end the best tree T_i* with the minimum validation error on L_V is selected from the sequence.

  A generic way of building the sequence of trees (iterating on the tree) is **reduced error pruning**. Reduced error pruning removes the node that most decreases the validation error at each step. The Classification and Regression trees algorithm introduces cost-complexity pruning [4] which defines a cost-complexity criterion:

  $$err(L_T, T) + \alpha \cdot Complexity(T)$$

  and builds the sequence of trees that minimize this criterion for increasing α.

3.3.2 Pre-pruning: Stopping Criteria

When to stop the decision tree learning algorithm is a tricky issue. The decision tree constructed using (almost perfectly classifying) all training set is overly fitting to the set and is less likely to give good results on the future data.

One method that we design is to stop when the maximum decrease in the impurity measure (heuristic) is smaller a threshold [14]. This method, also called 'forward pruning', has had mixed results. As it is explained in the CART algorithm a threshold $\beta > 0$ is set and the node t is declared terminal (leaf) if

$$\max_{s \in S} \Delta I(s, t) < \beta \tag{3.1}$$

where s is a split from the set of all possible splits S.

Another common method is adopting a maximum value for the depth of the tree in order to avoid over classifying the instances. We call the method as *maximum depth* stopping criterion. Thus, the depth of the tree at any branch can not exceed a number d. The authors of the decision tree algorithm, CART (Breiman et al. [4]) arrives at an opposite conclusions with the other leading decision tree algorithm, C4.5 (Quinlan [18]). Breiman et al. suggest 'pruning instead of stopping', also called 'backward pruning'. The maximum tree is grown having in mind some careful stopping criterion (lightly pre-pruned) in order to avoid extreme and unnecessary iterations and pruned upward. Breiman et al. constructs the maximum tree by continuing splitting until each terminal node

either is pure such that all instances have the same class ,
or is small such that the number of instances is less than a pre-specified number,
or contains only identical attribute-vectors (in which case splitting is impossible).

Even if the tree is going to be pruned backward the splitting procedure can be stopped when the tree is sufficiently large. The details will be given in the next Section 3.3.3.

Estimated Node Size

Instead of a pre-specified number we try an adaptive version of the method presented in the Equation 3.1. We estimate the branching factor $\bar{b}$ by calculating the average of the branching factors used so far and compute the expected number of matching instances to the node as

$$\mathbb{E}[N(t)] = \frac{N}{\bar{b}^d}$$

where N is the total number of instances used for training. Thus, each terminal node is either *pure* or matches less than $\alpha(d) \cdot \mathbb{E}[N(t)]$ number of instances

$$N(t) \leq \alpha(d) \cdot \frac{N}{\bar{b}^d}$$

where $\alpha(d)$ depends in the depth d of the node.

3.3.3 Backward Pruning: Minimal Cost-complexity Tree Pruning

The CART methodology involves two quite separate calculations. First the optimum value of the cost penalty for the tree complexity $\alpha*$ is determined using a data set which is independent from the data set used to train the tree. The second step is using this optimum value $\alpha*$ to select the final best pruned tree. In the first step CART algorithm uses a **pruning process** in order to get the 'right size' of a tree and accurate estimates of the true probabilities of misclassification. We present the first step of the algorithm in Figure 3.3.3. This pruning process which is the search for the 'right-sized' tree involves pruning or collapsing some of the branches of the largest tree (T_{max}) from the bottom up. During the pruning process the cost complexity parameter and an independent data sample from the training data is used to measure the predictive accuracy of the pruned tree. The pruning process produces a series of sequentially nested sub trees, each tree with two types of misclassification costs, i.e. the evaluation cost generated from an independent sample and the re-substitution cost generated from the learning sample, and the cost-complexity-parameter value. The evaluation cost can be generated using cross-validation or a test sample both of which should be independent from the learning sample [24].

MINIMAL COST-COMPLEXITY PRUNING($\mathbf{X}, k$)

1 **Input:**
  $\mathbf{X}$: a set of N labeled instances
  k : maximum number of trees
2 $T_{max} \leftarrow$ LEARNER($\mathbf{X}$, **attrs**, **0**)
3 $T_1 = T(\alpha_{min})$ where $\alpha_{min} = 0$:
  $R(T_1) = R(T_{max})$
4 **for** $i \leftarrow 1$ **to** k
  do
5 **for** $\forall t \in T_i$ $g_i(t) = \begin{cases} \frac{R(t)-R(T_t)}{|\widetilde{T}_t|-1}, & \text{if } t \notin \widetilde{T}_i \\ \infty, & \text{else} \end{cases}$
6 Choose the weakest link $\bar{t}_i$:
    $\bar{t}_i = \arg\min_{t \in T_i} g_i(t)$
    $g_i(\bar{t}_i) = \min_{t \in T_i} g_i(t)$
  and the set of weakest links $\{\bar{t}_i\}$
    $\{\bar{t}_i\} = \{\bar{t}'_i \colon g_i(\bar{t}_i) = g_i(\bar{t}'_i)\}$
7 $\alpha_{i+1} \leftarrow g_i(\bar{t}_i)$
8 $T_{i+1} \leftarrow T_i - T_{\bar{t}_i}$, $\forall \bar{t}_i \in \{\bar{t}_i\}$
9 **return** *The Sequence of Pruned Trees and their Complexity parameters*
  $T_1 \succ T_2 \succ \ldots \succ T_k$ and $\{\alpha_i\} : \alpha_i < \alpha_{i+1}$, $k \geq 1$, $\alpha_1 = 0$
  $T(\alpha_i) = T_i$, for $i \geq 1$, $\alpha_i \leq \alpha < \alpha_{i+1}$.

Figure 3-5: Algorithm: Minimal Cost-Complexity Pruning

Pruning to a Sequence of Trees

Breiman [4, 63] suggests growing a sufficiently large initial tree $T_{\max}$ by specifying a number $N_{\min}$ and continues splitting until each terminal node is either *pure* or contains only identical instances or matches less than $N_{\min}$ number of instances

$$N(t) \leq N_{\min}.$$

Instead of setting a constant number we adopt the heuristic described above 3.2 and compute the expected number of matching instances to the node. Thus, the decision tree is grown until each terminal node is either *pure* or matches less than $\alpha \cdot \mathbb{E}[N(t)]$ number of instances

$$N(t) \leq \alpha \cdot \frac{N}{\overline{b}^d}$$

where α is 5% since we want to eliminate outliers and the number of matching instances should not drop less than 5% of the expected number.

Starting from the maximum tree T_{max} the pruning process produces a finite sequence of subtrees T_1, T_2, ... with progressively fewer terminal nodes. Considering the maximum tree T_{max} there is a great number of subtrees and many distinct ways of pruning up to the root node. Thus, a 'selective' pruning procedure is necessary i.e. a selection of a reasonable number of subtrees. Each selected subtree which is the 'best' of its size range will be smaller in size than the previous one.

To evaluate how good a subtree T is its misclassification cost $R(T)$ on the training set is used. Although $R(T)$ lacks as an estimation of the population misclassification cost $R^*(T)$ it is the most natural criterion to use in comparing different subtrees of the same size. Moreover, there is a trade-off between accuracy and complexity of a decision tree starting from the most complex tree with low or zero misclassification to the simplest tree (the trivial tree containing of one node) with a very high misclassification [4]. Thus, the CART algorithm defines the **cost-complexity** measure of a decision tree which is the linear combination of the misclassification error of the tree and its complexity. For any subtree T having the same root as the maximum tree T_{max} ($T \preceq T_{max}$) the **cost-complexity measure** $R_\alpha(T)$ is defined as

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}|$$

where the complexity of the subtree $|\widetilde{T}|$ is the number of terminal nodes in T and the complexity parameter $\alpha \geq 0$ is a real number. The complexity parameter α which is the complexity cost per terminal node can also be understood as a cost penalty for the complexity of the decision tree.
If $\alpha = 0$, then cost complexity attains its minimum for the largest possible tree. On the other hand, as α increases and is sufficiently large (say, infinity), a tree with one terminal node (the root node) will have the lowest cost complexity. As values of α decrease and approach zero, trees that minimize cost complexity become larger. The 'right-sized' tree with 'correct' complexity should lie between these two extremes. Breiman et al. discuss how to estimate α and offer a detailed account of the pruning process [4].

For every value of α, there exists the **smallest minimizing subtree** $T(\alpha)$ defined by the conditions

$$\begin{aligned} R_\alpha(T(\alpha)) &= \min_{T \preceq T_{max}} R_\alpha(T) \\ \text{If } R_\alpha(T) &= R_\alpha(T(\alpha)), \text{ then } T(\alpha) \preceq T. \end{aligned}$$

Although α increases as continuous values there are at most finite number of subtrees of T_{max}. Due to finiteness, even if $T(\alpha)$ is the minimizing tree for a given value of α it will continue to be the minimizing until a 'jump point' α' is reached as the values of α increases. Then a new tree $T(\alpha')$ becomes the minimizing until the next 'jump point' α''.

Starting with $\alpha = 0$ cost complexity attains its minimum for the largest possible tree. That is, $T_1 = T(0)$ is the smallest subtree of T_{max} satisfying

$$R(T_1) = R(T_{max})$$

Thus, any terminal node of T_{max} whose misclassification error adds up to the same value with their immediate ancestor node t must be pruned off to get T_1 from T_{max}. T_{max} is pruned until no more pruning is possible. After obtaining T_1 the weakest-link cutting process is applied at each iteration. For any node $t \in T_i$

$$R_\alpha(t) = R(t) + \alpha$$

For the branch T_t containing the node t as the root node

$$R_\alpha(T_t) = R(T_t) + \alpha|\widetilde{T_t}|$$

The branch T_t has a smaller cost-complexity than the single node t if

$$R_\alpha(T_t) < R_\alpha(t)$$

At some critical value of α when the two cost-complexities become equal the single node t as a

branch would be preferable to the branch T_t since it is smaller than T_t. The critical value of α can be calculated when

$$\begin{aligned} R_\alpha(T_t) &\leq R_\alpha(t) \\ \alpha &\leq \frac{R(t) - R(T_t)}{|\widetilde{T}_t| - 1} \end{aligned}$$

Therefore, we define the function $g_i(t)$, $t \in T_i$ by

$$g_i(t) = \begin{cases} \frac{R(t)-R(T_t)}{|\widetilde{T}_t|-1}, & \text{if } t \notin \widetilde{T}_i \\ \infty, & \text{else} \end{cases}$$

where $\bar{t_i}$ is the weakest link because it is the first node where its cost-complexity, $R_\alpha(t)$, becomes equal to the cost-complexity, $R_\alpha(T_t)$, of its branch, T_t, as the parameter α naturally increases. Thus, $\bar{t_i}$ becomes preferable to $T_{\bar{t_i}}$ the value of α_{i+1} at which the equality occurs.

$$\begin{aligned} \alpha_{i+1} &= g_i(\bar{t_i}) \\ g_i(\bar{t_i}) &= \min_{t \in T_i} g_i(t) \end{aligned}$$

The next tree T_{i+1} is created by pruning away the branch $T_{\bar{t_i}}$ from the current tree T_i that is

$$T_{i+1} = T_i - T_{\bar{t_i}}.$$

If at any iteration there is a multiplicity of weakest links such that

$$g_i(\bar{t_i}) = g_i(\bar{t_i'})$$

where $\bar{t_i} \neq \bar{t_i'}$ then the next tree is created by pruning away any branch that the weakest list constructs.

$$T_{i+1} = T_i - T_{\bar{t_i}} - T_{\bar{t_i'}}$$

The pruning process continues like that until there is the root node t_0 left which is the trivial tree and we obtain a decreasing sequence of subtrees

$$T_{max} \succ T_1 \succ T_2 \succ \ldots \succ t_0$$

In the meantime we obtain an increasing sequence of complexity parameters $\{\alpha_i\}$ that is $\alpha_i < \alpha_{i+1}$, $k \geq 1$, where $\alpha_1 = 0$. For $i \geq 1$, $\alpha_i \leq \alpha < \alpha_{i+1}$, $T(\alpha) = T(\alpha_i) = T_i$.

Selecting the Best Pruned Subtree: An Estimation Problem

The best way to test the predictive accuracy of a tree is to take an independent test data set with known class distributions and run it down the tree and determine the proportion of cases misclassified. In some cases, such a data set is impossible due the available number of data. Considering this difficulty, Breiman et al. [4] provide three procedures for estimating the accuracy of tree-structured classifiers.

Re-substitution, Test Sample and Cross-validation. The training error, i.e. re-substitution error, can be used to select the best tree. However, the training error is biased and it is not a good way to evaluate the generalization error. To achieve an unbiased evaluation of the decision trees it

requires a second test. The test set is used as a validation set to evaluate the generalization error. In the case of missing a test set one part of the learning set needs to be dedicated as the validation set this may cause a problem if the learning set is a small set. A solution for that problem is using cross-validation Figure 3.3.3.

k-fold Cross-Validation(L, φ)

1 **Input:**
  L : a set of N labeled instances
  φ : Decision tree learning algorithm
2 $L_1, L_2, \ldots, L_k : \sum_{i=1}^{k} L_i = L,\ |L_i| = \frac{|L|}{k}\ \forall i$
3 **for** $i \leftarrow 1$ **to** k
  do
4 Generate a tree φ_i with the training set $L - L_i$
5 Evaluate the tree with the validation set L_i
    k_i : the number of misclassified instances
6 $R_i = err(\varphi_i, L_i) = \frac{k_i}{|L_i|}$
7 **return** *k-fold cross-validation error estimate*
  $err(\varphi, L) = \frac{1}{k} \sum_{i=1}^{K} err(\varphi_i, L_i)$

Figure 3-6: Algorithm: k-fold Cross-Validation

Let $L_1, L_2, \ldots, L_k$ be the sub-samples which are randomly split the training set L into k subsets of an equal number of observations. For each sub-sample a decision tree φ_i is generated with the training set $L - L_i$ and evaluated with the validation set L_i. Let k_i be the number of misclassified instances from the validation set. The error estimate R_i is determined by the proportion of misclassified observations

$$err(\varphi_i, L_i) = \frac{k_i}{|L_i|}$$

As a result we obtain a series of test sample re-substitution estimates, $err(\varphi_1, L_1)$, $err(\varphi_2, L_2)$,.., $err(\varphi_K, L_K)$. Thus, the k-fold cross-validation error estimate $err(\varphi, L)$ defined by

$$err(\varphi, L) = \sum_{i=1}^{K} \frac{|L_i|}{|L|} \cdot err(\varphi_i, L_i) = \frac{1}{k} \sum_{i=1}^{K} err(\varphi_i, L_i)$$

Cross-validation and cost-complexity pruning is combined to select the value of α. First, the trees within the sequences are matched up, based on their number of terminal nodes, to produce an estimate of the performance of the tree in classifying a new independent dataset, as a function of the number of terminal nodes or complexity. The method is to estimate the expected error rates of estimates obtained with φ for all values of α using cross-validation. The value $\alpha*$ is that value of α which minimizes the mean cross-validation error estimate and this is the estimated true error rate of $\varphi_{\alpha*}$ [14]. Once $\varphi_{\alpha*}$ has been determined, the tree that is finally suggested for use is that which minimizes the cost-complexity using $\varphi_{\alpha*}$ for all the data. Cross-validation estimation allows a data-based estimate of the tree complexity which results in the best performance with respect to an independent dataset. Using this method, a minimum cost occurs when the tree is complex enough to fit the information in the learning dataset, but not so complex that "noise" in the data is fit.

3.4 Evaluation of Decision Trees

Over-fitting is a very serious problem for a classifier. When the classifier is trained using a complex predictor on too-few examples it ends up over-fitting the too-few examples and it is not good enough to classify the future data any more. In order to avoid the over-fitting the trees should be evaluated using a different data set than the training data set. The generic way is using an extra data set to test the classifier. However, for moderate-sized samples, we use the cross-validation [14]. Basically cross-validation consists of dividing the data into sub-samples. We use each sub-sample to test the classifier constructed from the remaining sub-samples, and the estimated error rate is the average error rate from these sub-samples. In this way the error rate is estimated efficiently and in an unbiased way. Practically the use of cross-validation is the k-fold repetition of the learning cycle, which requires much computational effort. We select 10 folds for the cross-validation. Decreasing the number of folds will likely decrease the amount of time it takes for the decision tree to be generated, and increasing the number of folds will likely increase the amount of time it takes. Increasing the number of folds will create a larger dataset for the training data, which may increase accuracy of the decision tree.

For example, we select the tree with the best classification results on the test set from the multiple trees constructed by the Multiple-Decision Tree Builders, i.e., Bagging and Boosting. This decreases the over-fitting but it results in worse classification results than the actual classifier.

Chapter 4

Implementation of G01.0 in Drools

The diagram in the Figure 4-1 explains the overview picture of the system. The Drools WorkingMemory which is a StatefullSession is the main input of the Tree factory. Besides WorkingMemory, there are the parameters as input to the algorithm in order to build the decision tree. The most important parameter is the target object class. We construct a structured schema from the target object class. The other important parameters are

- *Splitting criterion heuristic function*: Information gain, information gain ratio
- *List of Stopping Criterion*:
  - node size estimation:
  - maximum depth
  - impurity improvement
- *Tree Builder Types*
  - Single tree builder
  - Forest builder: The user informs the number of trees to *bagg*.
  - Ada Boost builder: The user informs the number of trees to *boost*.
- Error Estimation for Tree pruner
  - Test Sample Estimation
  - Cross validation

The output of the tree factory is a decision tree. Rule Printer adapts the decision tree to the Drools' Parser by generating the valid rules.

4.1 Decision Tree Factory

The diagram in the Figure 4-2 explains the Tree Factory component. First, we create the Memory from the Drools's Working Memory. Memory contains the Instances, which are list of instances assignable from the target object class, and the Structured Schema, which defines the target class object. We construct the Structured Schema by visiting the target class and its parent classes, and their attributes and the attributes' parent classes. Thus, we get all simple attributes related to the target class.

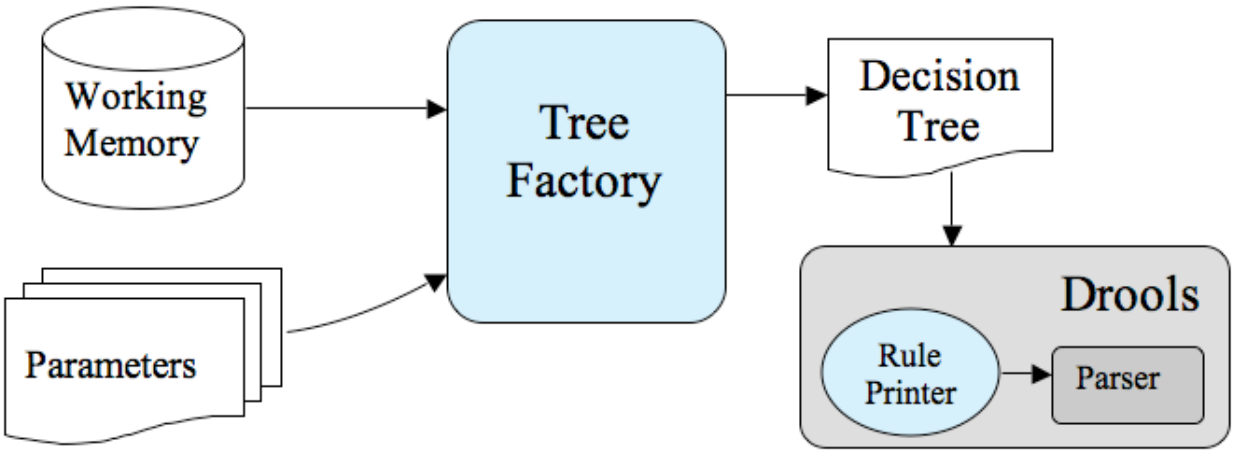

Figure 4-1: The Overview Diagram of the System

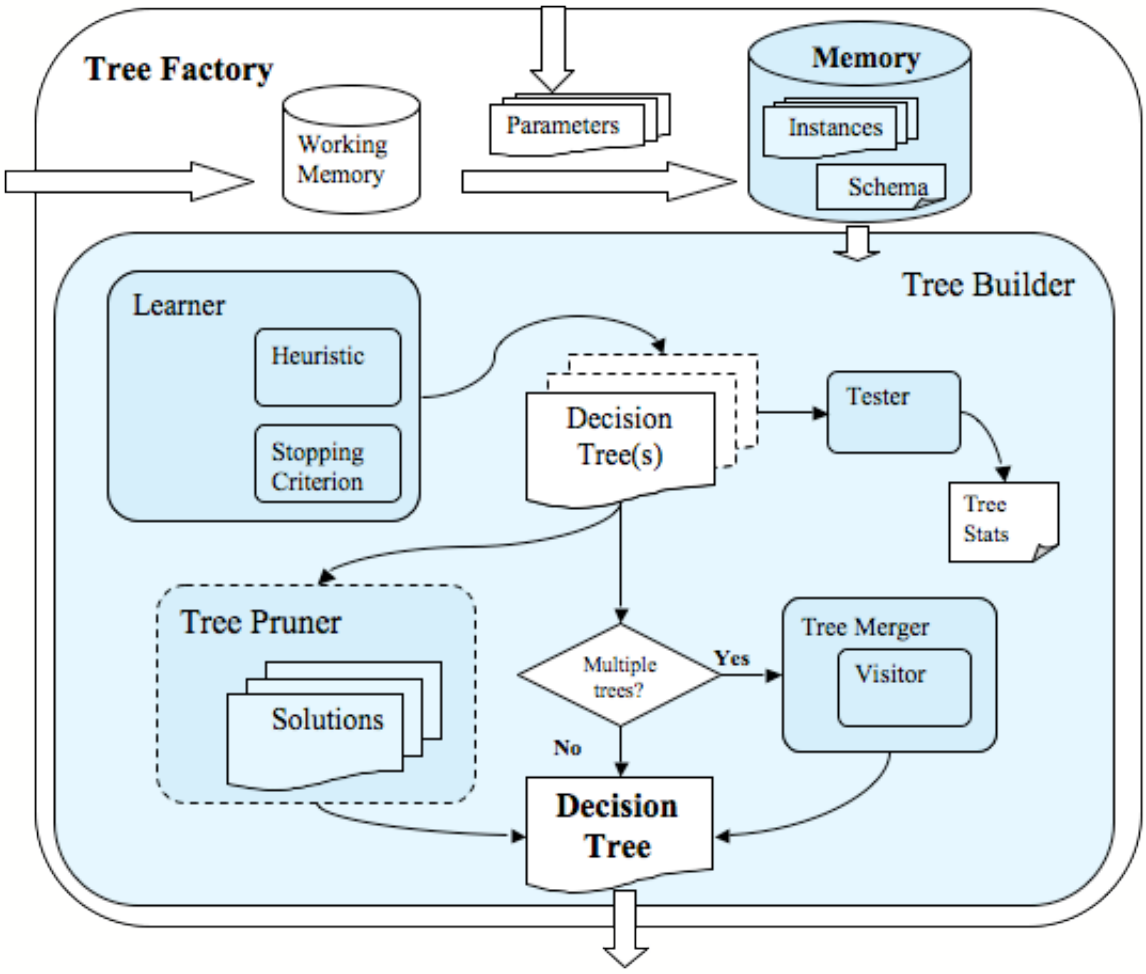

Figure 4-2: The Tree Factory

4.1.1 Feeding Objects, Building Schema and Instance List

We create an instance from each object in the Working Memory and we add the instance to the Memory. The objects coming from the Working Memory can contain attributes with Complex domains which have different type of types than the classes we have seen. For each none existent attribute type we create a new domain in the set of domains. Then we insert the instance into its instance list as a new instance.

- *Memory*: the memory object used for the training and the testing decision trees
  - session: Drools' Working Memory
  - targetClass: the Java class of the target Objects
  - instances: Hash map of *instances list*s referenced by their class

  DomainSet: Hash table of the attribute domains referenced by the names of all distinct attributes defined by the valid class.
- *Instance List*: the instances which belongs to the same class constructs the instance lists
  - schema: The specifications of the class
  - instance factory: the factory which can create instances using the Drools' Working memory and the schema. It instantiates the attributes of any object from the session if the class of that schema is assignable from the class of the object.
  - validDomains: Hash table of the attribute domains referenced by the attribute names that belong to the class.

    Memory.insert(Object element): We create a new instance from the attributes of element. We add each attribute of the object to instance with its domain specifications. If the value of that attribute does not exist in the domain we add the value to the possible values of the domain.
- *Instance*: Instance has its attributes
  - fields: Hash table of the attribute domains referenced by the attribute names that belong to the object which created the fact
  - values: Hash table of the attribute values referenced by the attribute names that belong to the object which created the fact

4.1.2 How to Build Decision Trees

The Tree Builder is the main responsible for populating decision trees. It trains as many decision trees as necessary using Learner with the heuristic function and the stopping criterion. As a result of training, we get a single tree in the case of Single Tree Builder or multiple trees in the case of Tree Forests, AdaBoost, or AdaBoostK. If the user chooses to prune the decision tree(s) Tree Pruner computes the optimum error estimation by pruning each decision tree to a sequence of decision trees and then select the best decision tree using the optimal estimate. In any case we compute the training and the testing error using the Tree Tester which depends on the tree builder type and then we save the results of the testing as tree performance statistics. If Decision Tree pruner is not used and multiple trees are constructed we merge the trees into one tree using the Tree Visitor to evaluate the rule performances of the trees. However, the final tree loses classification accuracy when it is a merge of multiple trees. As a result, the final decision tree is the output of the Tree Factory.

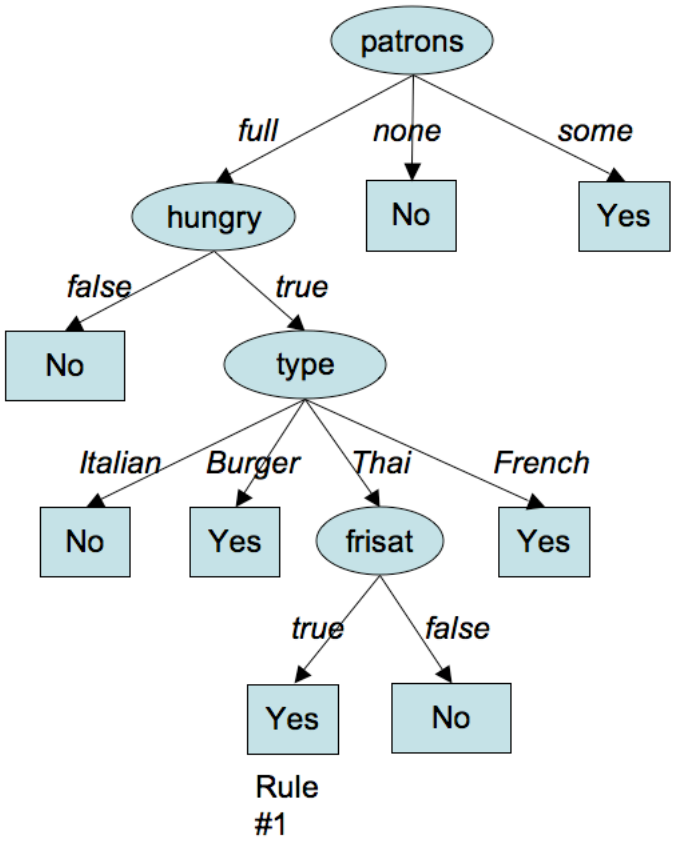

Figure 4-3: The Decision Tree Result of Restaurant Objects

4.2 From a Tree to Rules

We use a Tree Visitor to construct the rules from the decision tree. Tree Visitor uses Depth First Search algorithm to visit all possible branches of the tree. It processes Declarations, Conditions, and Actions of each rules to get the structured rule. The output rules depend on the learning algorithm used to train the tree.

Decision Trees from ID3 Algorithm

We implemented the Quinlan's ID3 Algorithm. We compared the results with two other ID3 implementations. One is the Java implementation of ID3 algorithms from Online Code Repository [15] of the AIMA book [22]. The functions give the same results i.e. entropy function and it computes the same trees.

For example: Using the 12 Restaurant objects with 12 discrete attributes (6 boolean, 3 literal and 1 numerical) and Boolean target attribute the ID3 implementation constructs the decision tree given in the Figure 4-3. This decision tree contains the rules given in the Section 5.2.1. After visiting this decision tree the Rule Printer prints the rules in the DRL file Figure 5-3.

Decision Trees from C4.5 Algorithm

We implemented almost all features introduced in C.45 Algorithm [18]. We compared the results using Quinlan's data, i.e. 15 Golf objects with 4 attributes (1 boolean, 1 literal and 2 numerical)

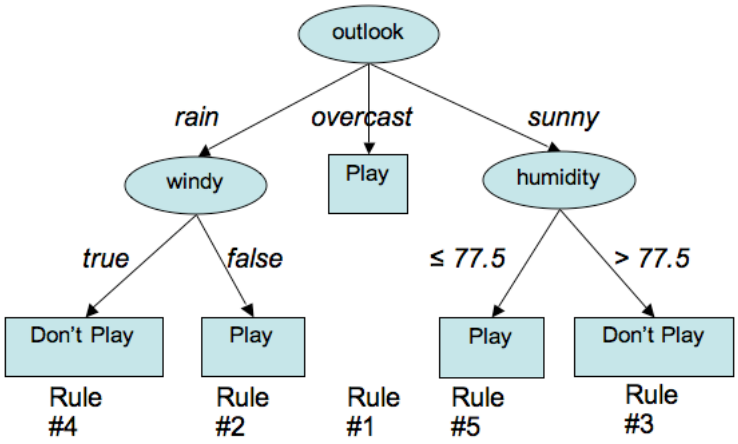

Figure 4-4: The Decision Tree Result of Golf Objects

and Boolean target attribute. We describe the Golf objects in the Section 5.1.2. The C4.5 implementation constructs the decision tree Figure 4-4. This decision tree contains the rules given in the Section 5.2.2. After visiting this decision tree the Rule Printer prints the rules in the DRL file Figure 5-9.

4.3 Decision Tree Pruner Details

When Decision Tree Pruner uses the cross validation during the Decision Tree Forward Pruning it builds an entire tree and prunes it to the sequence of trees k times, i.e. k-fold Cross Validation. Thus, there are k sequences of trees produced. This procedure is very costly considering a huge number of instances with many number of attributes. To speed up the procedure we store extra information at each tree node; specifically, what the highest represented class is and what proportion of instances at the node actually are that class. This helps the pruner to know what would happen if it pruned that branch and replaced it with a leaf; the leaf would be labeled with the highest represented class. Thus, we can easily recalculate the change in the training error, and the new training error. This saves us from calculating the training error by dropping the all instances to the tree every time we will choose the branch to prune. This optimization can be called Online Testing. Moreover, it can be applied to the test set, too, which would improve the Decision Tree Forward Pruning algorithm a lot.

4.3.1 Annotations: Interaction with the User

The algorithm has to assign one of the attributes as the target. The user annotates the field using Field Annotations 4.3.1. For example:

```
@FieldAnnotation(readingSeq = 1, target = true, discrete = false)
private double y;
```

Moreover, the user can define her own labeling function. To achieve this she writes a getter function in any object class related to the target object class. She annotates the getter function using the Class Annotations 4.3.1. For example, here the **getLabel()** function given in the Program 2 in the Section 5.1.6 is annotated in the Triangle class:

```
@ClassAnnotation(label_element = "getLabel")
```

```
public class Triangle {
    ......
}
```

The attributes are assumed to be discrete by default. If the user specifies the opposite using the Field annotations then the domain is treated as continuous. On the other hand, if the number of possible values of the domain is more than a number then the domain is treated as continuous in order to improve the learning algorithm. The whole specifications of the attribute domains are read from the Annotations given at the fields of the object classes. Thus, we can validate the type of the input with respect to the annotated field type. We can also validate the value of the field given the possible set of values in the annotation.

- Field Annotations:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
public @interface FieldAnnotation {
    int readingSeq() default 0;
    boolean ignore() default false;
    boolean skip() default false;
    boolean target() default false;
    boolean discrete() default true;
    String[] values() default {"x"};
}
```

- Class Annotations:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface ClassAnnotation {
    String label_element() default "";
}
```

4.3.2 Multi-Threaded ID3 Algorithm

The ID3 learning algorithm can be paralleled with two different approaches.

1. The simple approach is to assign a thread to each branch coming from the tree node. The number of branched is the number of possible values of the chosen attribute at the tree node.

2. The other approach is to assign a thread to each possible attribute during the process of choosing the attribute that will branch the tree node so the loop on the attributes can be eliminated.

We tried the simple approach. It is a very easy and simple way of parallelizing. It is very easy to parallel the construction of the decision tree until the branches are merged due to the joins between different object. Only caveat is that the 'putNode(Object attributeValue, TreeNode newNode)' function which assigns the children of the nodes should be protected.

The second approach needs more attention because the threads must join after calculating the entropy value of their attribute. Then the attribute with the greatest information gain will be selected so the execution will continue with the chosen attribute.

Chapter 5

Experimental Evaluation

5.1 Data

5.1.1 Restaurant

Restaurant objects first used in the AIMA book [22]. The ID3 implementation of the AIMA Online Code Repository The ID3 implementation from Online Code Repository [15] of the AIMA book [22] uses the 12 Restaurant objects with 11 discrete attributes (7 boolean, and 4 literal) and Boolean target attribute, i.e., will_wait. The Table 5.1 explains the detailed structure of the restaurant objects. Since the Restaurant object structure does not have any continuous attribute the ID3 algorithm is enough to train the decision tree classifying the restaurant objects.

5.1.2 Golf

Golf objects first used in the Quinlan's book [18]. The data set contains 15 Golf objects with 4 attributes (1 boolean, 1 literal and 2 numerical) and Boolean target attribute, i.e., decision. The Table 5.2 explains the detailed structure of the golf objects. Since the Golf object structure has continuous attributes the C4.5 algorithm is required to train the decision tree to generate rules as to when to play, and when not to play, a game of golf.

5.1.3 Cars

Car Evaluation Database was derived from a simple hierarchical decision model originally developed for the demonstration of DEX [2] [1]. The model evaluates cars according to the concept structure given in the Figure 5-1 and the Table 5.3. The Car Evaluation Database contains examples with the structural information removed, i.e., directly relates CAR to the six input attributes: buying, maint, doors, persons, lug_boot, safety. There are 1728 instances and 6 attributes. The instances completely cover the attribute space and there are no missing attribute values. The Table 5.4 gives the number of instances per class, i.e. Class Distribution.

5.1.4 Nursery

Nursery Database was derived from the same model within expert system shell for decision making DEX [2] like Car Evaluation Database explained in the previous Section [1]. The model was originally developed to rank applications for nursery schools. It was used during several years in 1980's when there was excessive enrollment to these schools in Ljubljana, Slovenia, and the rejected

Table 5.1: The Structure of the Restaurant Database

attribute	**values**	**domain type**
will_wait	Yes, No.	boolean
alternate	Yes, No.	boolean
bar	Yes, No.	boolean
fri_sat	Yes, No.	boolean
hungry	Yes, No.	boolean
patrons	None, Some, Full	discrete
price	$, $$, $$$	discrete
raining	Yes, No.	boolean
reservation	Yes, No.	boolean
type	French, Italian, Thai, Burger	discrete
wait_estimate	'0-10','10-30','30-60','>60'	discrete

Table 5.2: The Structure of the Golf Database

attribute	**values**	**domain type**	**information**
decision	Play, Don't Play.	binary	target attribute
outlook	sunny, overcast, rain	discrete	the weather conditions
temperature	-	continuous	integer values between
humidity	-	continuous	integer values between
windy	true, false	binary	if wind exists

Table 5.3: The Concept Structure of the Car Evaluation Database

Structure				**Attribute Values**	**Explanation**
CAR*				unacc, acc, good, v-good	car acceptability
.	PRICE				overall price
.	.	buying		v-high, high, med, low	buying price
.	.	maint		v-high, high, med, low	price of the maintenance
.	TECH				technical characteristics
.	.	COMFORT			comfort
.	.	.	doors	2, 3, 4, 5-more	number of doors
.	.	.	persons	2, 4, more	capacity, i.e. number of persons
.	.	.	lug_boot	small, med, big	the size of luggage boot
.	.	safety		low, med, high	estimated safety of the car

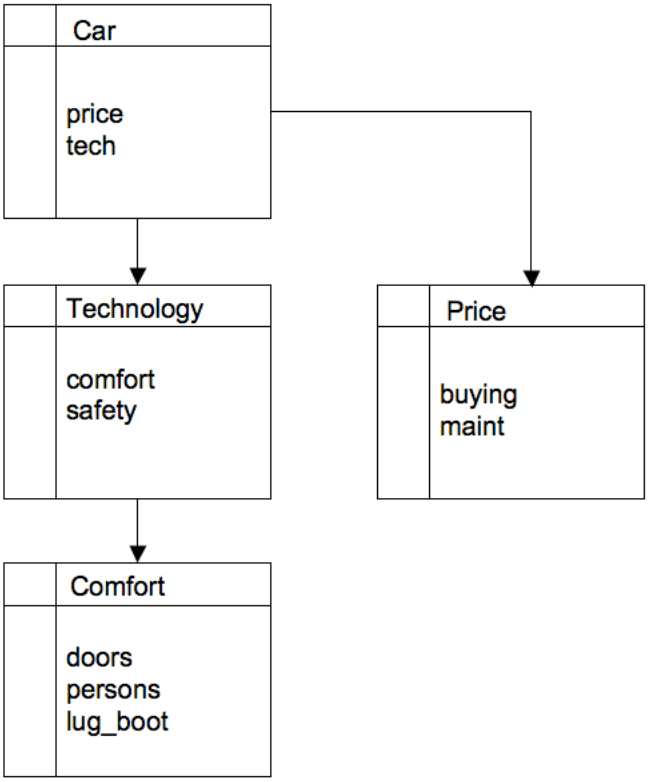

Figure 5-1: The Concept Structure of the Car Object

Table 5.4: The Class Distribution of the Car Evaluation Database

class	**N**	**N[%]**
unacc	1210	(70.023 %)
acc	384	(22.222 %)
good	69	(3.993 %)
v-good	65	(3.762 %)

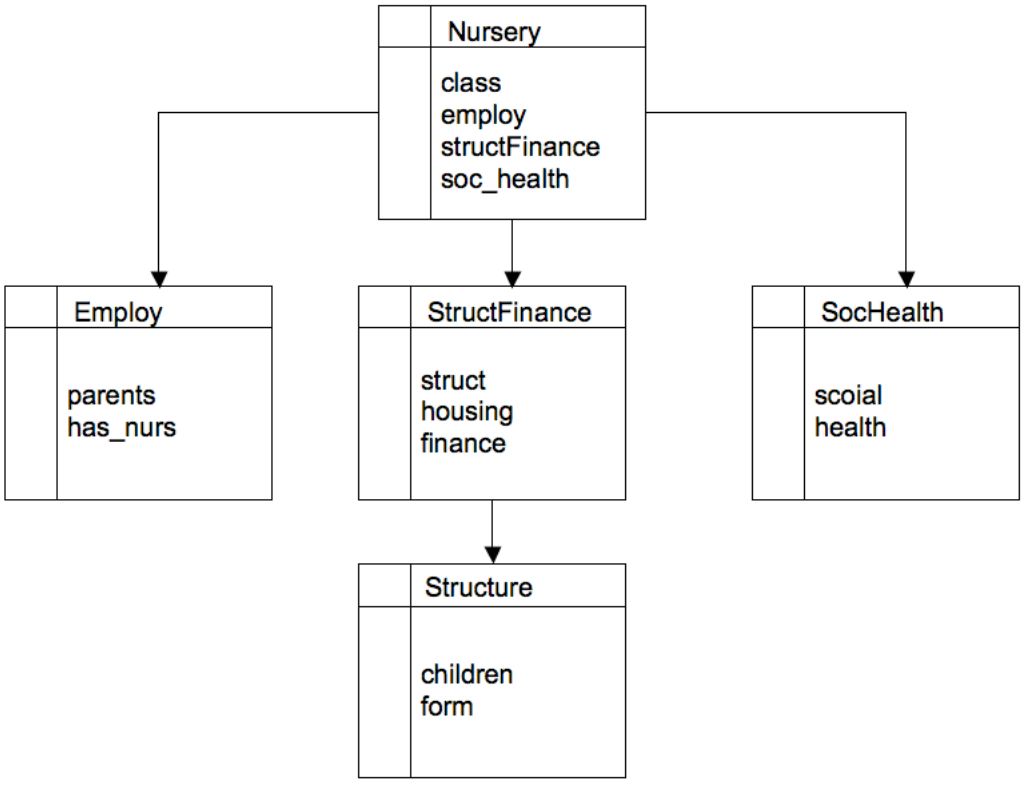

Figure 5-2: The Concept Structure of the Nursery Object

applications frequently needed an objective explanation. The final decision depended on three subproblems: occupation of parents and child's nursery, family structure and financial standing, and social and health picture of the family, i.e., directly relates NURSERY to the eight input attributes: parents, has_nurs, form, children, housing, finance, social, and health, respectively. The hierarchical model ranks nursery-school applications according to the concept structure given in the Figure 5-2 and the Table 5.5. Input attributes are printed in lowercase. Besides the target concept (NURSERY) the model includes four intermediate concepts: EMPLOY, STRUCT_FINAN, STRUCTURE, SOC_HEALTH.

The Nursery Database contains 12960 instances and 8 attributes. The instances completely cover the attribute space and there are no missing attribute values. The Table 5.6 gives the number of instances per class, i.e. Class Distribution.

5.1.5 Poker Hands

Poker hands database is used first by [5] [1]. Each record is an example of a hand consisting of five playing cards drawn from a standard deck of 52. Each card is described using two attributes (suit and rank), for a total of 10 predictive attributes given in the Table 5.7. There is one Class attribute that describes the 'Poker Hand'. The order of cards is important, which is why there are 480 possible Royal Flush hands as compared to 4 (one for each suit). There are 311875200 records possible in the entire domain even if only 2598960 different poker hands exist. However, 1,025,010 records exists in the database i.e. 25010 records available for training, and 1,000,000 for testing. Each record has 10 predictive attributes, 1 goal attribute. There are no missing attribute values. The Table 5.8 gives the statistics of the entire domain, i.e., number of instances per class.

Table 5.5: The Concept Structure of the Nursery Database

Structure	**Attribute Values**	**Explanation**
NURSERY*	not_recom, recommend, very_recom, priority, spec_priord	Evaluation of applications for nursery schools
. EMPLOY		Employment of parents and child's nursery
. . parents	usual, pretentious, great_pret	Parents' occupation
. . has_nurs	proper, less_proper, improper, critical, very_crit	Child's nursery
. STRUCT_FINAN		Family structure and financial standings
. . STRUCTURE		Family structure
. . . form	complete, completed, incomplete, foster	Form of the family
. . . children	1, 2, 3, more	Number of children
. . housing	convenient, less_conv, critical	Housing conditions
. . finance	convenient, inconv	Financial standing of the family
. SOC_HEALTH		Social and health picture of the family
. . social	non-prob, slightly_prob, problematic	Social conditions
. . health	recommended, priority, not_recom	Health conditions

Table 5.6: The Class Distribution of the Nursery Database

class	**N**	**N[%]**
not_recom	4320	(33.333 %)
recommend	2	(0.015 %)
very_recom	328	(2.531 %)
priority	4266	(32.917 %)
spec_prior	4044	(31.204 %)

Table 5.7: The Structure of the Poker Hands Database

attribute	values	domain type	information
CLASS	(0-9)	ordinal	poker hand
S1	(1-4)	Numerical	Suit of card #1 representing
		Numerical	Hearts, Spades, Diamonds, Clubs
C1	(1-13)	Numerical	Rank of card #1 representing
		Numerical	(Ace, 2, 3, ... , Queen, King)
S2	(1-4)	Numerical	Suit of card #2 representing
		Numerical	Hearts, Spades, Diamonds, Clubs
C2	(1-13)	Numerical	Rank of card #2 representing
		Numerical	(Ace, 2, 3, ... , Queen, King)
S3	(1-4)	Numerical	Suit of card #3 representing
		Numerical	Hearts, Spades, Diamonds, Clubs
C3	(1-13)	Numerical	Rank of card #3 representing
		Numerical	(Ace, 2, 3, ... , Queen, King)
S4	(1-4)	Numerical	Suit of card #4 representing
		Numerical	Hearts, Spades, Diamonds, Clubs
C4	(1-13)	Numerical	Rank of card #4 representing
		Numerical	(Ace, 2, 3, ... , Queen, King)
S5	(1-4)	Numerical	Suit of card #5 representing
		Numerical	Hearts, Spades, Diamonds, Clubs
C5	(1-13)	Numerical	Rank of card #5 representing
		Numerical	(Ace, 2, 3, ... , Queen, King)

Table 5.8: The Class Distribution of the Poker Hands Database

class		# of hands	N	N[%]	information
0	Nothing in hand	1302540	156304800	(50.117739 %)	not a recognized poker hand
1	One pair	1098240	131788800	(42.256903 %)	one pair of equal ranks
2	Two pairs	123552	14826240	(4.753902 %)	two pairs of equal ranks
3	Three of a kind	54912	6589440	(2.112845 %)	three equal ranks
4	Straight	10200	1224000	(0.392464 %)	five cards, sequentially ranked with no gaps
5	Flush	5108	612960	(0.196540 %)	five cards with the same suit
6	Full house	3744	449280	(0.144058 %)	pair + different rank three of a kind
7	Four of a kind	624	74880	(0.024010 %)	four equal ranks
8	Straight flush	36	4320	(0.001385 %)	straight + flush
9	Royal flush	4	480	(0.000154 %)	Ace, King, Queen, Jack, Ten + flush

Table 5.9: The Structure of the Triangle Database

attribute	values	domain type	information
label	true, false	binary	target attribute
x	$[0, 10.0)$	continuous	the length of an edge
y	$[0, 10.0)$	continuous	the length of an edge
z	$[0, 10.0)$	continuous	the length of an edge

Table 5.10: The Class Distribution of the Triangle Database

class	N	N[%]
true	5007	(50.07 %)
false	4993	(49.93 %)

5.1.6 Triangle

Triangle Database contains 10000 randomly generated triangle candidates. Triangles candidates has 3 continuous attributes, i.e. the length of the edges, and a target attribute specifying if the object is a valid triangle or not. The Table 5.2 explains the details of the attributes. The Table 5.10 gives the statistics of the entire domain, i.e., number of instances per class.

Triangles are labeled according to the triangle rule, i.e., the sum of every two sides of a triangle must be greater than the third side. There is the **getLabel()** function given in the Program 2 that returns the label of each triangle candidate object.

Program 2 The User-Defined Labeling Function: **getLabel()**

```
public boolean getLabel() {
    return (z < x + y) && (x < z + y) && (y < x + z);
}
```

5.2 The Rules out of Decision Trees

5.2.1 Training with ID3 Algorithm: Restaurant Database

When we train the decision tree with the ID3 algorithm we can use only the discrete attributes to split the instances at each node. Thus, the conditions of the rules can contain only equality constraints, i.e., $hungry == true$. Using ID3 algorithm we construct the decision tree which gives 8 rules:

Rule #1 suggests that
if $patrons == Full$ **and** $hungry == true$ **and** $type == Italian$
then Willing to Wait is No

Rule #2 suggests that

if $patrons == Full$ **and** $hungry == true$ **and** $type == Burger$
then Willing to Wait is Yes

Rule #3 suggests that
if $patrons == Full$ **and** $hungry == true$ **and** $type == Thai$ **and** $frisat == true$
then Willing to Wait is Yes

Rule #4 suggests that
if $patrons == Full$ **and** $hungry == true$ **and** $type == Thai$ **and** $frisat == false$
then Willing to Wait is No

Rule #5 suggests that
if $patrons == Full$ **and** $hungry == true$ **and** $type == French$
then Willing to Wait is Yes

Rule #6 suggests that
if $patrons == Full$ **and** $hungry == false$
then Willing to Wait is No

Rule #7 suggests that
if $patrons == None$
then Willing to Wait is No

Rule #8 suggests that
if $patrons == Some$
then Willing to Wait is Yes

5.2.2 Training with C4.5 Algorithm: Golf Database

When we train the decision tree with the C4.5 algorithm we can use both type attributes, i.e. discrete and continues, to split the instances at each node. Thus, the conditions of the rules can contain only equality, i.e., $outlook == rain$ or inequality, i.e., $humidity > 77.5$ constraints. Using C4.5 algorithm we construct the decision tree which gives 5 rules:

Rule #1 suggests that
if $outlook == overcast$
then $Play$

Rule #2 suggests that
if $outlook == rain$ **and** $windy == false$
then $Play$

Rule #3 suggests that
if $outlook == sunny$ **and** $humidity > 77.5$
then $DonotPlay$

Rule #4 suggests that
if $outlook == rain$ **and** $windy == true$
then $DonotPlay$

Rule #5 suggests that
if $outlook == sunny$ **and** $humidity \leq 77.5$
then $Play$

5.3 Comparison of Decision Tree Learners

The Learner uses all available data for training the decision trees so only we present the training errors. The Learners discretize the attributes with quantitative domains using a recursive discretization method applicable at any tree node and the discretization recursion stops at a constant depth.

We measure the quality of the RETE Network by its number of nodes, i.e., ObjectTypeNodes, AlphaNodes, JoinNodes, and TerminalNodes. We explain these different type of nodes in the Section 2.0.3. These statistics of RETE Network classifying the different databases are in the Table 5.12. The comparison of the classification results of each decision tree belonging to the RETE Network is in the Table 5.11.

5.3.1 ID3 Decision Tree Learning Algorithm

5.3.2 Restaurants

The rules as a result of the decision tree are in the DRL file in the Figure 5-3. Drools constructs the RETE Tree from the decision tree in the Figure 5-4.

Car

The decision tree produced the 188 rules; however, only the first three best rule whose classification rank is significantly high (is bigger than 0.05) given in the DRL file in the Figure 5-5. Since there are too many number of rules produced from the decision tree we select the first best three rule in the Figure 5-6. The classification results are in the Table 5.11. We compare the number of nodes of the RETE Tree in the Table 5.12.

Nursery

The decision tree produced the 839 structured rules; however, only the first best 8 rules whose classification rank is significantly high (is bigger than 0.02) given in the DRL file in the Figure 5-7. Since there are too many number of rules produced from the decision tree we select the first best 8 rules in the Figure 5-8. The classification results are in the Table 5.11. We compare the number of nodes of the RETE Tree in the Table 5.12.

5.3.3 C4.5 Decision Tree Learning Algorithm

Golf

And the decision tree produces the rules in the DRL file in the Figure 5-9. The classification results are compared in the Table 5.11. This drl file can be parsed by the Drools and we get the Rete tree in the Figure 5-10.

```
package examples.learner;
import examples.learner.Restaurant

rule "#7 will_wait= true  classifying 3.0 num of facts with rank:0.2727272727272727"
     when
         $restaurant_0 : Restaurant(patrons == "Some", $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), " +
         "Classified as (true )");
end
rule "#1 will_wait= false  classifying 2.0 num of facts with rank:0.18181818181818182"
     when
         $restaurant_0 : Restaurant(patrons == "None", $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), "+
         "Classified as (false )");
end
rule "#4 will_wait= false  classifying 2.0 num of facts with rank:0.18181818181818182"
     when
         $restaurant_0 : Restaurant(patrons == "Full", hungry == false,
                                    $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), "+
         "Classified as (false )");
end
rule "#2 will_wait= false  classifying 1.0 num of facts with rank:0.09090909090909091"
     when
         $restaurant_0 : Restaurant(patrons == "Full", hungry == true,type == "Italian",
         $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), "+
         "Classified as (false )");
end
rule "#3 will_wait= false  classifying 1.0 num of facts with rank:0.09090909090909091"
     when
         $restaurant_0 : Restaurant(patrons == "Full", hungry == true, type == "Thai",
         fri_sat == false, $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), "+
         "Classified as (false )");
end
rule "#5 will_wait= true  classifying 1.0 num of facts with rank:0.09090909090909091"
     when
         $restaurant_0 : Restaurant(patrons == "Full", hungry == true, type == "Thai",
         fri_sat == true, $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), "+
         "Classified as (true )");
end
rule "#6 will_wait= true  classifying 1.0 num of facts with rank:0.09090909090909091"
     when
         $restaurant_0 : Restaurant(patrons == "Full", hungry == true,type == "Burger",
         $target_label : will_wait )
     then
         System.out.println("[will_wait] Expected value (" + $target_label + "), "+
         "Classified as (true )");
end
```

Figure 5-3: The DRL file for Restaurant

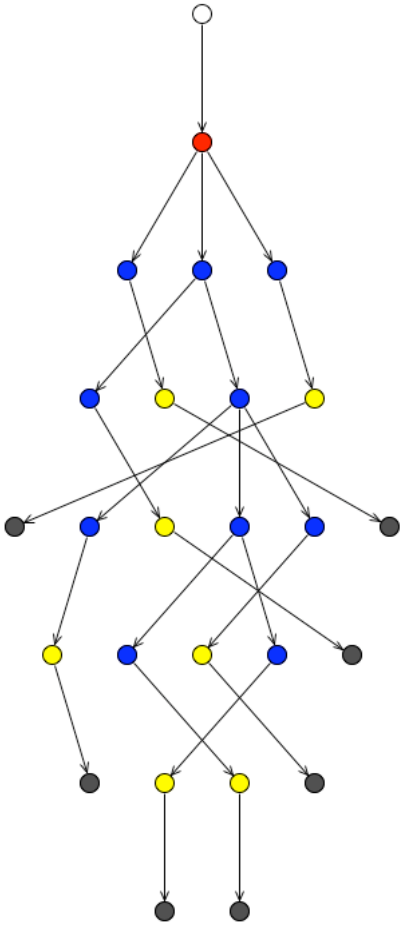

Figure 5-4: The RETE Network of Restaurant Object, Figure 5-3

```
package examples.learner;

import examples.learner.Car

rule "#131 target= unacc  classifying 576.0 num of facts with rank:0.3333333333333333"
     when
         $car_0 : Car(safety == "low", $target_label : target )
     then
         System.out.println("[target] Expected value (" + $target_label + "), Classified
as (unacc )");
end

rule "#59 target= unacc  classifying 192.0 num of facts with rank:0.1111111111111111"
     when
         $car_0 : Car(safety == "med", persons == "2", $target_label : target )
     then
         System.out.println("[target] Expected value (" + $target_label + "), Classified
as (unacc )");
end

rule "#140 target= unacc  classifying 192.0 num of facts with rank:0.1111111111111111"
     when
         $car_0 : Car(safety == "high", persons == "2", $target_label : target )
     then
         System.out.println("[target] Expected value (" + $target_label + "), Classified
as (unacc )");
end
```

Figure 5-5: The DRL file for Simple Car Object

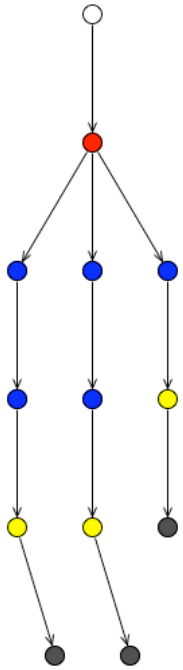

Figure 5-6: The RETE Network of Car Object, Figure 5-5

Triangle

Triangle problem is hardest case for decision trees as we can see from the classification results of the objects in the Table 5.11. Even if the decision tree contains only 7 rules presented with the DRL file in the Figure 5-11 the training error, i.e. 20.22 %, is the biggest of all data sets. The RETE Network belonging to the decision tree is in the Figure 5-12. As we can see from the best rule classifying 3667 facts, i.e., 36.67 % of the facts since rank of the rule is 0.3667, the instances do not homogeneously cover the domain and the rule suggests that if all sides are bigger than 2.5658 ± 0.3662 then the object is a valid triangle.

```
rule "#6 label= true  classifying 3667.0 num of facts with rank:0.3667"
    when
        $triangle_0 : Triangle(x > 2.19965, z > 2.9319499999999996, y > 2.1881,
                               $target_label : label )
    then
        System.out.println("[label] Expected value (" + $target_label + "),"+
                           " Classified as (true )");
end
```

Poker Hands

The decision tree produces 9383 rules, only 9247 of which classify at least one instance. The classification results are in the Table 5.11.

```
package examples.learner;

rule "#838 classnursery = not_recom classifying 4320 num of facts with rank:0.33"
     when
         Nursery(health == "not_recom", classnursery : classnursery)
     then
         System.out.println("Decision on classnursery = "+classnursery +": (not_recom)");
end
rule "#709 classnursery = priority classifying 288 num of facts with rank:0.022"
     when
         Nursery(health == "priority", has_nurs == "proper", parents == "pretentious",
                     classnursery : classnursery)
     then
         System.out.println("Decision on classnursery = "+classnursery +": (priority)");
end
rule "#710 classnursery = priority classifying 288 num of facts with rank:0.022"
     when
         Nursery(health == "priority", has_nurs == "proper", parents == "usual",
                     classnursery : classnursery)
     then
         System.out.println("Decision on classnursery = "+classnursery +": (priority)");
end
rule "#749 classnursery = priority classifying 288 num of facts with rank:0.022"
     when
         Nursery(health == "priority", has_nurs == "improper", parents == "usual",
                     classnursery : classnursery)
     then
         System.out.println("Decision on classnursery = "+classnursery +": (priority)");
end
rule "#778 classnursery = priority classifying 288 num of facts with rank:0.022"
     when
         Nursery(health == "priority", has_nurs == "less_proper", parents ==
"pretentious",
                     classnursery : classnursery)
     then
         System.out.println("Decision on classnursery = "+classnursery +": (priority)");
end
rule "#779 classnursery = priority classifying 288 num of facts with rank:0.022"
     when
         Nursery(health == "priority", has_nurs == "less_proper", parents == "usual",
                     classnursery : classnursery)
     then
         System.out.println("Decision on classnursery = "+classnursery +": (priority)");
end
```

Figure 5-7: The DRL file for Simple Nursery Object

Table 5.11: The Classification Results of the Databases, Restaurant, Golf, Car, Nursery, Triangle, and Poker hands

Database	**incorrect**		**correct**		**sum**	
.	**N**	**N[%]**	**N**	**N[%]**	**N**	**N[%]**
Restaurant	0	0.000	11	100.00	11	100.00
Golf	0	0.000	14	100.00	14	100.00
Car	0	0.000	1728	100.00	1728	100.00
Nursery	0	0.000	12960	100.00	12960	100.00
Triangle	2022	20.220	7978	79.780	10000	100.00
Poker Hands	2765	11.055	22245	88.937	25010	100.00

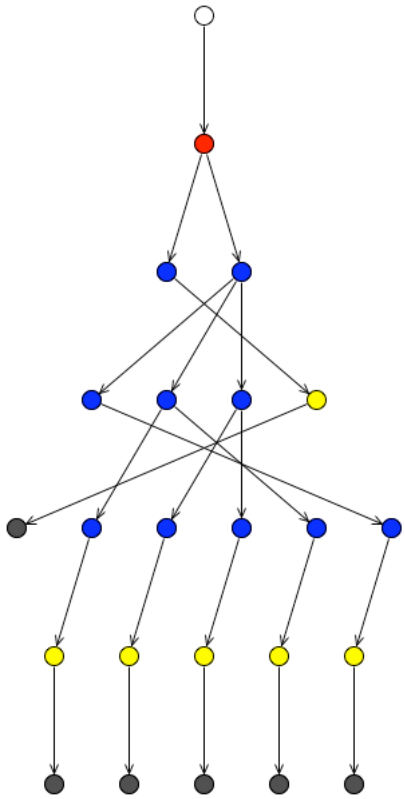

Figure 5-8: The RETE Network of Simple Nursery Object, Figure 5-7

Table 5.12: The RETE Network Statistics of the Databases, Restaurant, Golf, Car, Nursery, Triangle, and Poker hands

Database	**ObjectType**	**Alpha**	**Join**	**Terminal**
Restaurant	1	10	0	7
Golf	1	7	0	5
Car	1	278	0	188
Nursery	1	381	0	252
Triangle	1	12	0	7
Poker Hands	1	17577	0	9247

```
package examples.learner;
import examples.learner.Golf

rule "#0 decision= Play  classifying 4.0 num of facts with rank:0.2857142857142857"
     when
         $golf_0 : Golf(outlook == "overcast", $target_label : decision )
     then
         System.out.println("[decision] Expected value (" + $target_label + "),"+
         " Classified as (Play )");
end
rule "#3 decision= Play  classifying 3.0 num of facts with rank:0.21428571428571427"
     when
         $golf_0 : Golf(outlook == "rain", windy == false, $target_label : decision )
     then
         System.out.println("[decision] Expected value (" + $target_label + "),"+
         " Classified as (Play )");
end
rule "#4 decision= Don't Play  classifying 3.0 num of facts with rank:
0.21428571428571427"
     when
         $golf_0 : Golf(outlook == "sunny", humidity > 77, $target_label : decision )
     then
         System.out.println("[decision] Expected value (" + $target_label + "),"+
         " Classified as (Don't Play )");
end
rule "#1 decision= Play  classifying 2.0 num of facts with rank:0.14285714285714285"
     when
         $golf_0 : Golf(outlook == "sunny", humidity <= 77, $target_label : decision )
     then
         System.out.println("[decision] Expected value (" + $target_label + "),"+
         " Classified as (Play )");
end
rule "#2 decision= Don't Play  classifying 2.0 num of facts with rank:
0.14285714285714285"
     when
         $golf_0 : Golf(outlook == "rain", windy == true, $target_label : decision )
     then
         System.out.println("[decision] Expected value (" + $target_label + "),"+
         " Classified as (Don't Play )");
end
```

Figure 5-9: The DRL file for Golf Object

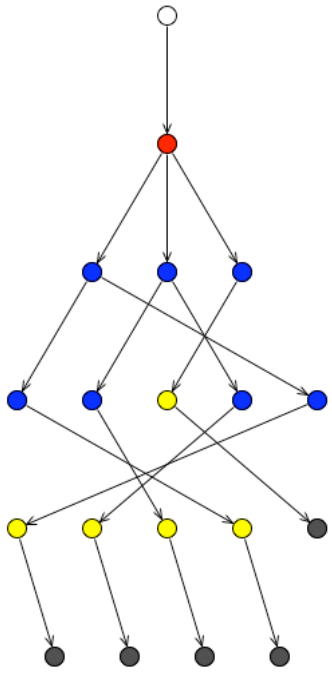

Figure 5-10: The RETE Network of Golf Object, Figure 5-9

```
package examples.learner;

import examples.learner.Triangle

rule "#6 label= true  classifying 3667.0 num of facts with rank:0.3667"
     when
         $triangle_0 : Triangle(x > 2.19965, z > 2.9319499999999996, y > 2.1881, $target_label :
label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (true )");
end
rule "#0 label= false  classifying 1570.0 num of facts with rank:0.157"
     when
         $triangle_0 : Triangle(x <= 2.19965, z > 0.27415, y > 0.7625, $target_label : label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (false )");
end
rule "#3 label= false  classifying 1280.0 num of facts with rank:0.128"
     when
         $triangle_0 : Triangle(x > 2.19965, z <= 2.9319499999999996, y > 1.70265, $target_label :
label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (false )");
end
rule "#2 label= false  classifying 888.0 num of facts with rank:0.0888"
     when
         $triangle_0 : Triangle(x > 2.19965, z > 2.9319499999999996, y <= 2.1881, $target_label :
label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (false )");
end
rule "#4 label= false  classifying 361.0 num of facts with rank:0.0361"
     when
         $triangle_0 : Triangle(x > 2.19965, z <= 2.9319499999999996, y <= 1.70265, $target_label :
label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (false )");
end
rule "#1 label= false  classifying 146.0 num of facts with rank:0.0146"
     when
         $triangle_0 : Triangle(x <= 2.19965, z > 0.27415, y <= 0.7625, $target_label : label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (false )");
end
rule "#5 label= false  classifying 66.0 num of facts with rank:0.0066"
     when
         $triangle_0 : Triangle(x <= 2.19965, z <= 0.27415, $target_label : label )
     then
         System.out.println("[label] Expected value (" + $target_label + "), Classified as (false )");
end
```

Figure 5-11: The DRL file for Triangle Object

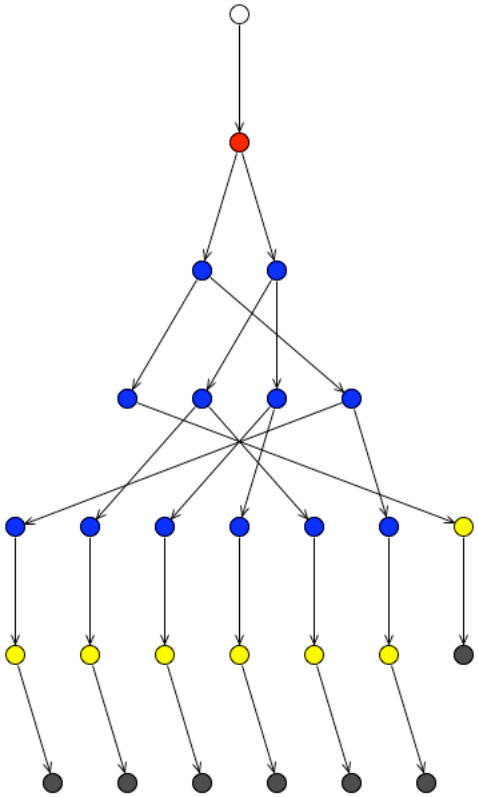

Figure 5-12: The RETE Network of Triangle Object, Figure 5-11

5.4 Comparison of RETE Network between Splitting Criteria Heuristic Functions

Here we compare the two different heuristics functions by the misclassification errors and the RETE Networks of the decision trees constructed using these heuristics. The heuristics functions are

- Entropy (Information Gain)
- Information Gain Ratio

Moreover, we used two other different Split criteria in order to see the affect of using a heuristic function. Here are the criterion

- Min Entropy: Selecting the attribute that gives the worst entropy value.
- Random: Selecting the attribute randomly.

We train the first two trees by optimizing the heuristic functions, i.e., Information Gain and Information Gain Ratio. On the other hand, we train the third tree by unoptimizing the Information Gain heuristic function and the last tree by randomly selecting the split attribute. As a result the learner of the first two decision trees always select the best attributes to split whereas the learner of the other decision trees select not so good attributes.

Figure 5-14 and Figure 5-16 compare the RETE Network statistics of the decision trees trained using these split criteria. The main classification objects are Poker (binary-class), Car and Nursery in the Figure 5-14, Poker (binary-class) and Poker (multi-class) in the Figure 5-16. Figure 5-13

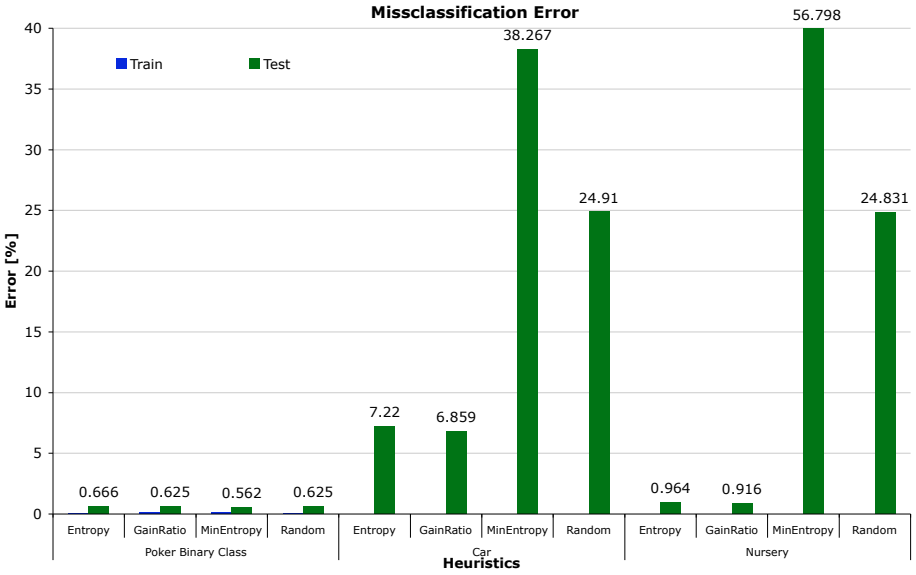

Figure 5-13: Misclassification Results on Poker (Binary), Car and Nursery database

and Figure 5-15 present the misclassification errors of each decision tree classifying the given main object. As we can see in the RETE statistic figures 5-14 and 5-16 all objects have been classified with almost 0 training error except the Poker (multi-class) objects. This is due to high complexity of the Poker (multi-class) objects. Gain Ratio heuristics results in slightly smaller test errors with all object types. However, Entropy function constructs smaller decision trees (less number of Alpha and Terminal nodes) with all object except the complicated case, i.e. the Poker (multi-class). the Poker (multi-class) object behaves completely opposite to the other objects: a lot more complex decision tree with less training error and more test error using Entropy heuristic function. Thus, we observe that two decision trees having very close misclassification error on the training and the test set can have different construction of the RETE Networks as in the case of Poker (binary-class), Car and Nursery objects.

Furthermore, the trees constructed without optimization ends up more than twice of the trees constructed using the optimization heuristic functions. MinEntropy and Random selection criteria construct a lot bigger RETE Networks even if they have bigger errors. You can see the misclassification error statistics in the Table 5.4 and the RETE Network statistics in the Table 5.4.

5.5 Multi-Relational Data Results

We compare the RETE Network statistics and classification results of the decision trees belonging to these the RETE Network between Structured and Unstructured Data. We use Car Evaluation and Nursery database for this purpose since they have the structure information. The measure for the tree quality is its number of nodes, i.e., ObjectTypeNodes, AlphaNodes, JoinNodes, and TerminalNodes. We explain these different type of nodes in the Section 2.0.3. Figure 5-17 compares RETE Network of simple Car object to its structured version and RETE Tree of simple Nursery object to its structured version. The RETE Network statistics are in the Table 5.16. The comparison of the classification results is in the Table 5.15. As we can observe from the results the misclassification errors are pretty same using the structured version objects but the RETE Network statistics differ a lot. The RETE Network statistics show that the structured rules have smaller number of AlphaNodes while the number of JoinNodes is increasing due the structured nature.

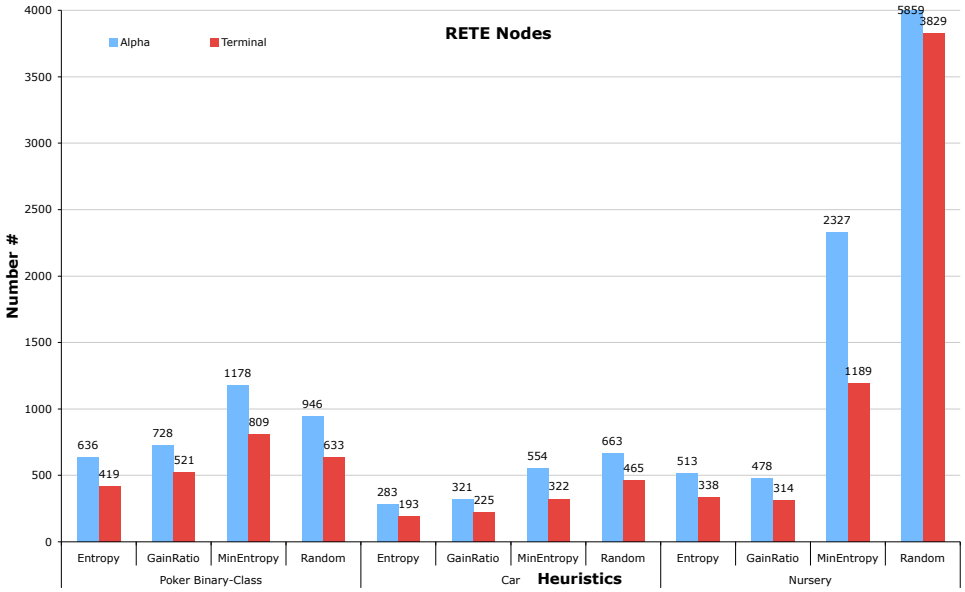

Figure 5-14: RETE Nodes on Poker (Binary), Car and Nursery database

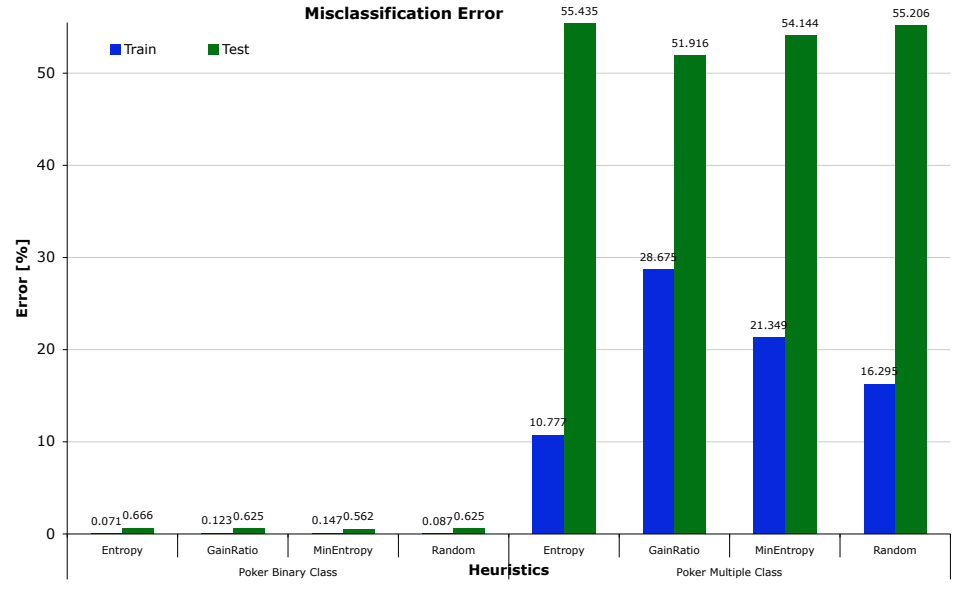

Figure 5-15: Misclassification Results on Poker Database, Binary v.s. Multi Class

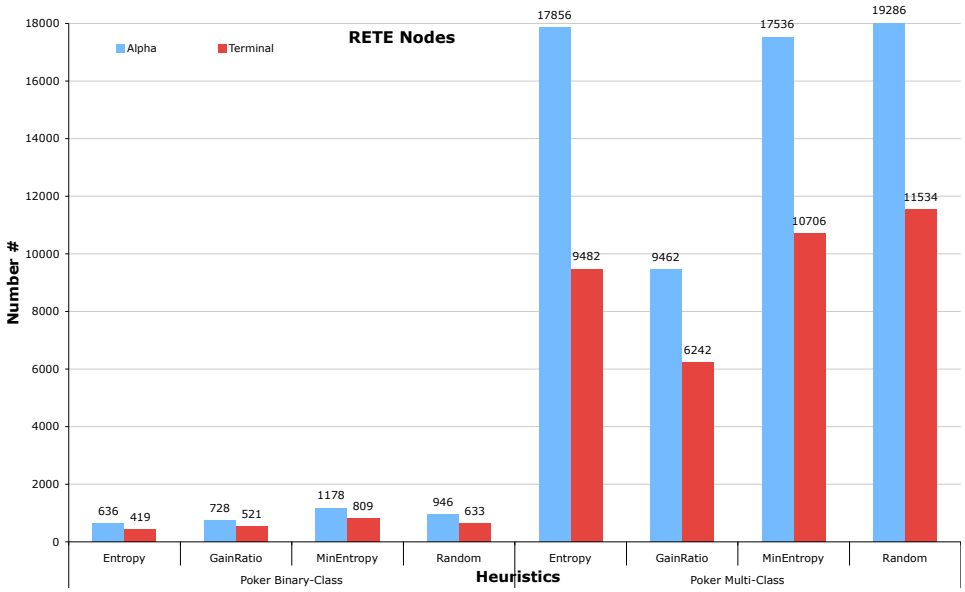

Figure 5-16: RETE Nodes on Poker Database, Binary v.s. Multi Class

		Training					Testing				
		incorrect		correct		total	incorrect		correct		total
Data	**Heuristic**	**N**	**N[%]**	**N**	**N[%]**	**N**	**N**	**N[%]**	**N**	**N[%]**	**N**
.	Entropy	18	0.071	25192	99.929	25210	32	0.666	4770	99.334	4802
Poker	GainRatio	31	0.123	25179	99.877	25210	30	0.625	4772	99.375	4802
Binary	MinEntropy	37	0.147	25173	99.853	25210	27	0.562	4775	99.438	4802
Class	Random	22	0.087	25188	99.913	25210	30	0.625	4772	99.375	4802
.	Entropy	2717	10.777	22493	89.223	25210	2662	55.435	2140	44.565	4802
Poker	GainRatio	7229	28.675	17981	71.325	25210	2493	51.916	2309	48.084	4802
Multiple	MinEntropy	5382	21.349	19828	78.651	25210	2600	54.144	2202	45.856	4802
Class	Random	4108	16.295	21102	83.705	25210	2651	55.206	2151	44.794	4802
.	Entropy	0	0	1451	100	1451	20	7.22	257	92.78	277
Car	GainRatio	0	0	1451	100	1451	19	6.859	258	93.141	277
.	MinEntropy	0	0	1451	100	1451	106	38.267	171	61.733	277
.	Random	0	0	1451	100	1451	69	24.91	208	75.09	277
.	Entropy	0	0	10886	100	10886	20	0.964	2054	99.036	2074
Nursery	GainRatio	0	0	10886	100	10886	19	0.916	2055	99.084	2074
.	MinEntropy	0	0	10886	100	10886	1178	56.798	896	43.202	2074
.	Random	0	0	10886	100	10886	515	24.831	1559	75.169	2074

Table 5.13: The Comparison of Classification Results on Training and Test Set between Splitting Criteria Heuristics

Data	Heuristic	ObjectType	Alpha	Join	Terminal
	Entropy	1	636	0	419
Poker	GainRatio	1	728	0	521
Binary	MinEntropy	1	1178	0	809
Class	Random	1	946	0	633
.	Entropy	1	17856	0	9482
Poker	GainRatio	1	9462	0	6242
Multiple	MinEntropy	1	17536	0	10706
Class	Random	1	19286	0	11534
.	Entropy	1	283	0	193
Car	GainRatio	1	321	0	225
.	MinEntropy	1	554	0	322
.	Random	1	663	0	465
	Entropy	1	513	0	338
Nursery	GainRatio	1	478	0	314
	MinEntropy	1	2327	0	1189
	Random	1	5859	0	3829

Table 5.14: The Comparison of the RETE Networks between Splitting Criteria Heuristics, Table 5.4

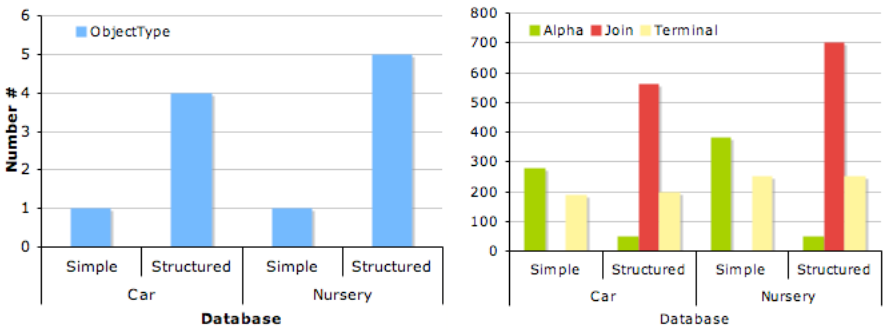

Figure 5-17: The Comparison of Rete Network Statistics between Structured and Simple Object Types, Car and Nursery

```
package examples.learner.structured_car;

import examples.learner.structured_car.Car
import examples.learner.structured_car.Tech
import examples.learner.structured_car.Comfort
import examples.learner.structured_car.Price

rule "#7 target= unacc  classifying 576.0 num of facts with rank:0.3333333333333333"
     when
         $tech_char_1 : Tech(safety == "low" )
         $car_0 : Car(tech_char == $tech_char_1, $target_label : target )
     then
         System.out.println("[target] Expected value (" + $target_label + "),"+"
Classified as (unacc )");
end

rule "#24 target= unacc  classifying 192.0 num of facts with rank:0.1111111111111111"
     when
         $comfort_2 : Comfort(persons == "2" )
         $tech_char_1 : Tech(safety == "high", comfort == $comfort_2 )
         $car_0 : Car(tech_char == $tech_char_1, $target_label : target )
     then
         System.out.println("[target] Expected value (" + $target_label + "),"+
         " Classified as (unacc )");
end

rule "#45 target= unacc  classifying 192.0 num of facts with rank:0.1111111111111111"
     when
         $comfort_2 : Comfort(persons == "2" )
         $tech_char_1 : Tech(safety == "med", comfort == $comfort_2 )
         $car_0 : Car(tech_char == $tech_char_1, $target_label : target )
     then
         System.out.println("[target] Expected value (" + $target_label + "),"+"
Classified as (unacc )");
end
```

Figure 5-18: The DRL file for Structured Car Object

Car

The decision tree produces the 197 structured rules; however, the best three rules whose classification rank is significantly high (is bigger than 0.05) given in the DRL file in the Figure 5-18.The RETE Network constructed from these selected rules are given in the Figure 5-19.

Nursery

As in the case of its simple object version the decision tree produces the 252 rules from structured Nursery object; however, the best 6 rules whose classification rank is significantly high (is bigger than 0.022) given in the DRL file in the Figure 5-20. The RETE Network constructed from these selected rules are given in the Figure 5-21

5.6 Decision Tree Post-Pruning Statistics

We present the results for the Decision Tree Pruner using Poker (binary-class) and the Nursery data. We do the post-pruning experiments using TestSample estimation.

First we give the statistics of each pruned tree classifying Poker (binary-class) data in the Figure 5-22. The first generated tree over-fits the data since the training error is zero and the test error is the maximum. When we prune the tree the training error is increasing while the test error is decreasing until it reaches the best point where the test error is minimum. The over-fitting is decreasing until the best point. After the best point the tree starts to under-fit the data since the tree is not complex enough, i.e., it is over-pruned.

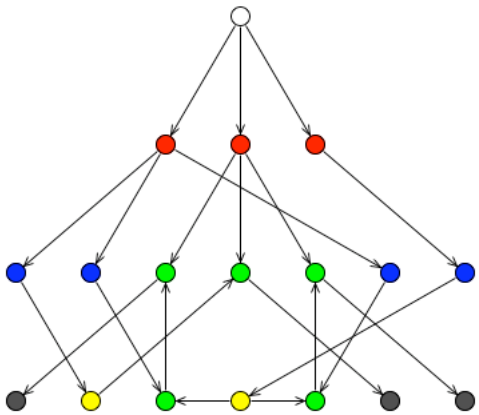

Figure 5-19: The RETE Network of the Structured Car Object, Figure 5-18

```
rule "#105 classnursery= not_recom  classifying 4320.0 num of facts with rank:0.33"
     when
         $soc_health_1 : SocHealth(health == "not_recom" )
         $nursery_0 : Nursery(soc_health == $soc_health_1, $target_label : classnursery )
     then
         System.out.println("[classnursery] Expected value (" + $target_label + "),"
                            +" Classified as (not_recom )");
end
rule "#4 classnursery= priority  classifying 288.0 num of facts with rank:0.022"
     when
         $employ_2 : Employ(has_nurs == "proper", parents == "pretentious" )
         $soc_health_1 : SocHealth(health == "priority" )
         $nursery_0 : Nursery(soc_health == $soc_health_1, employ == $employ_2,
                              $target_label : classnursery )
     then
         System.out.println("[classnursery] Expected value (" + $target_label + "),"
                            +" Classified as (priority )");
end
rule "#38 classnursery= priority  classifying 288.0 num of facts with rank:0.022"
     when
         $employ_2 : Employ(has_nurs == "proper", parents == "usual" )
         $soc_health_1 : SocHealth(health == "priority" )
         $nursery_0 : Nursery(soc_health == $soc_health_1, employ == $employ_2,
                              $target_label : classnursery )
     then
         System.out.println("[classnursery] Expected value (" + $target_label + "),"
                            +"Classified as (priority )");
end
rule "#121 classnursery= priority  classifying 288.0 num of facts with rank:0.022"
     when
         $employ_2 : Employ(has_nurs == "less_proper", parents == "pretentious" )
         $soc_health_1 : SocHealth(health == "priority" )
         $nursery_0 : Nursery(soc_health == $soc_health_1, employ == $employ_2,
                              $target_label : classnursery )
     then
         System.out.println("[classnursery] Expected value (" + $target_label + "),"
                              +"Classified as (priority )");
end
rule "#130 classnursery= priority  classifying 288.0 num of facts with rank:0.022"
     when
         $employ_2 : Employ(has_nurs == "improper", parents == "usual" )
         $soc_health_1 : SocHealth(health == "priority" )
         $nursery_0 : Nursery(soc_health == $soc_health_1, employ == $employ_2,
                              $target_label : classnursery )
     then
         System.out.println("[classnursery] Expected value (" + $target_label + "),"
         +"Classified as (priority )");
end
rule "#210 classnursery= priority  classifying 288.0 num of facts with rank:0.022"
     when
         $employ_2 : Employ(has_nurs == "less_proper", parents == "usual" )
         $soc_health_1 : SocHealth(health == "priority" )
         $nursery_0 : Nursery(soc_health == $soc_health_1, employ == $employ_2,
                              $target_label : classnursery )
     then
         System.out.println("[classnursery] Expected value (" + $target_label + "),"
         +"Classified as (priority )");
end
```

Figure 5-20: The DRL file for Structured Car Object

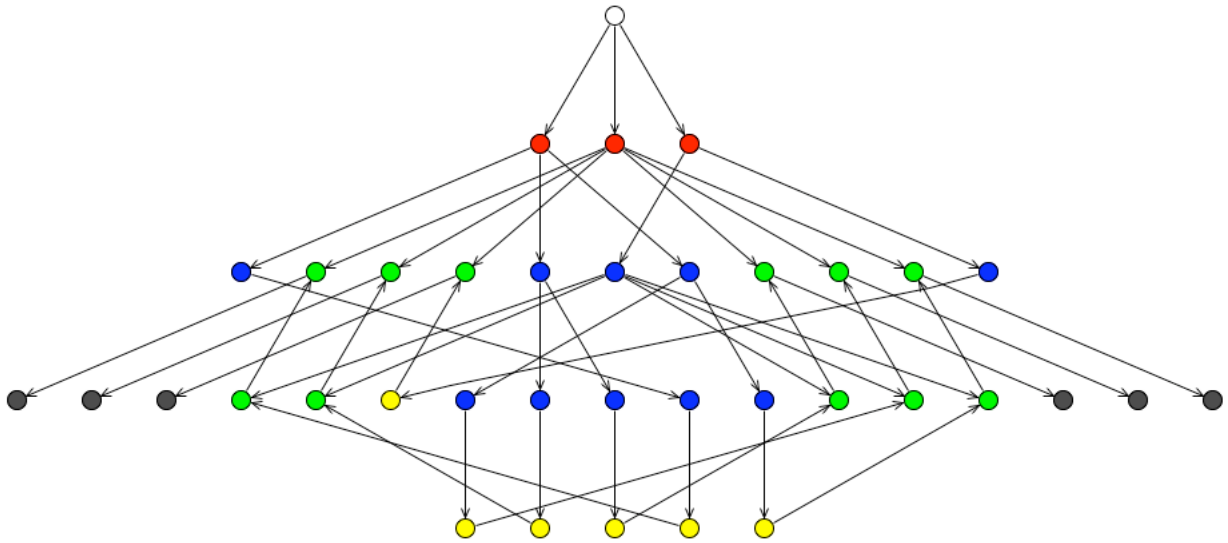

Figure 5-21: The RETE Network of the Structured Nursery Object, Figure 5-20

Table 5.15: The Comparison of Classification Results between Structured and Unstructured Data

Database	**incorrect**		**correct**		**sum**	
.	**N**	**N[%]**	**N**	**N[%]**	**N**	**N[%]**
Car	0	0.000	1728	100.00	1728	100.00
Structured Car	0	0.000	1728	100.00	1728	100.00
Nursery	0	0.000	12960	100.00	12960	100.00
Structured Nursery	0	0.000	10368	100.00	10368	100.00

Table 5.16: The Comparison of RETE Network Statistics between Structured and Unstructured Data

Database	**ObjectType**	**Alpha**	**Join**	**Terminal**
Car	1	278	0	188
Structured Car	4	49	564	197
Nursery	1	381	0	252
Structured Nursery	5	48	703	252

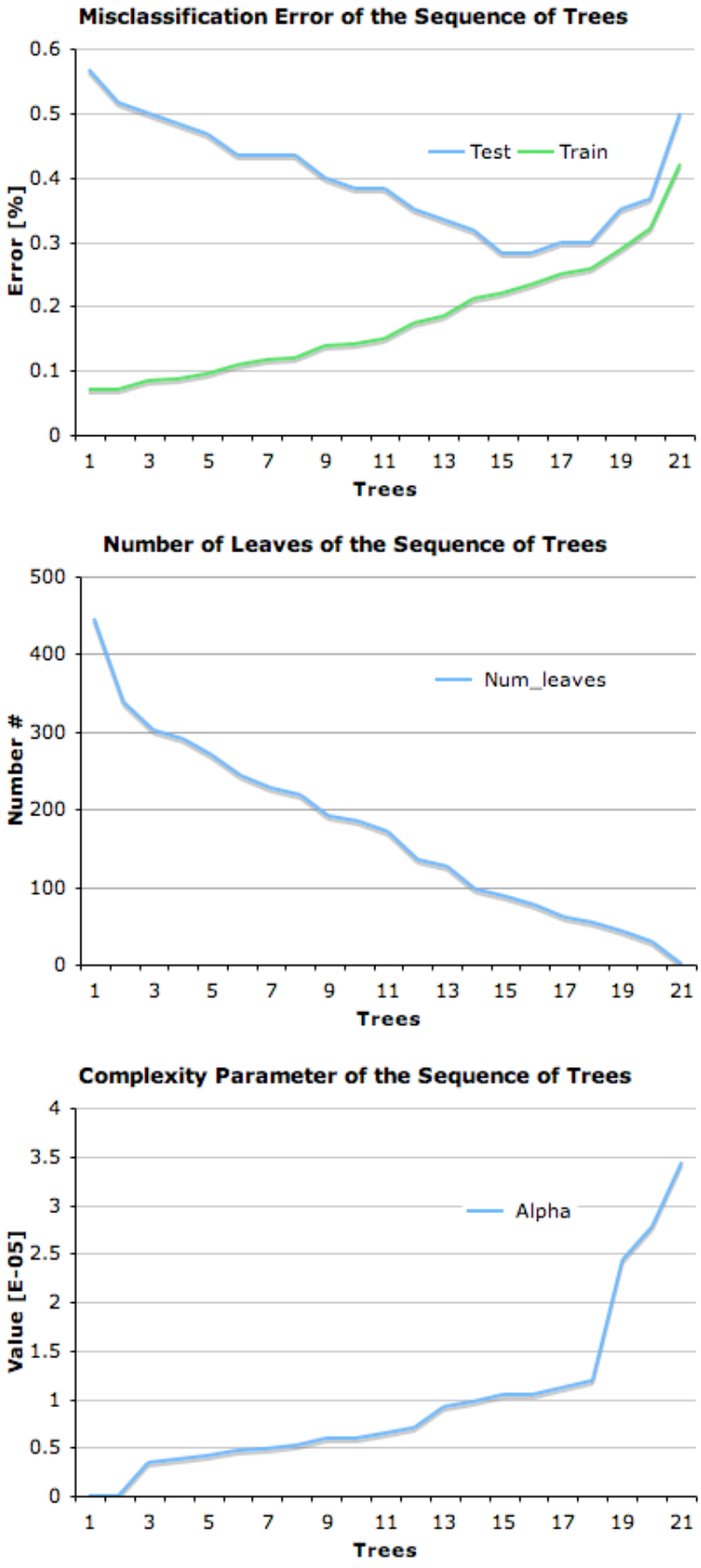

Figure 5-22: Minimal Cost-Complexity Pruning Statistics, Poker Database, Binary Classification

As we can observe from the decrease in the RETE size graph in the Figure 5-22 the pruning algorithm initially tends to prune off large branches with many terminal nodes. As the trees get smaller it tends to cut off fewer at a time. In other words, the iterations create sub trees with a decreasing complexity (a decreasing number of terminal nodes) and an increasing cost (re-substitution relative cost).

However, we give the statistics of each pruned tree classifying Nursery data in the Figure 5-23. The number of leaves and the complexity parameter changes as expected. The number of leaves decreases and the complexity parameter increases as a result of pruning. However, the test error which starts from almost zero increases every step. Thus, pruning does not always guarantee decrease in the test error as you can see in the Figure 5-23.

5.7 Comparison of Decision Tree Builders

Here we compare the classification results on training and test set between different builders, i.e., Single, Boosting, and, Bagging. The total number of data is 30012, and we use 84 % to train and the rest to test. We classify the Poker Hands objects. We use 10 trees for Boosting and Bagging. Since the AdaBoost algorithm works only with binary target attributes Poker Hand instances are labeled using an artificial target function. The target is to find if the poker hand is a good hand such that it is at least a Flush, i.e., five cards with the same suit. There is the **getLabel()** that returns the label of each poker instance.

```
public boolean getLabel() {
    return poker_hand>=5;
}
```

We did the experiments first without any type of pruning, neither pre-pruning nor post-pruning. Then we repeat the experiments with pre-pruning. We use two stopping criterion for pre-pruning. The first criteria is to prune if the number of matching instances is less than half of the estimated node size (Section 3.3.2). The other criteria is to prune if the depth of the branch is more than 70 % of the number of attributes. At last we do the experiments with post-pruning using TestSample estimation. We present the comparison of classification results on training and test set between builders in the Figure 5-25 and the numbers in the Table 5.17. The first group of data gives the trees without any pruning applied. The second group is the trees when there are pre-pruned. The data marked with a star (*) is the error of the classifier generated and the last on is the tree selected from the builder when it builds multiple trees.

We give the comparison of RETE Network statistics in the Figure 5-24 and the number in the Table 5.18. The statistics of how many times the decision trees are pruned using Estimated Node Size, Impurity Decrease and Maximum Depth is in the Table 5.19. For the bagging and the boosting the numbers are the sum of the multiple trees.

As we can see in the Figure 5-25 training errors are always smaller than the test error. This shows that the trees are likely to be over-fitting.

The Pre-Pruning does not give so good results as expected. The training and test error rates are high and the final Rete Networks are not that small compared to the builders without pruning. The Post-Pruning has more balanced error rates, i.e. their training and testing errors are closer to each other. Besides the classification errors, we can see the post-pruned trees have significantly less number of nodes than the other ones in the Figure 5-24.

The Multiple-Tree Builders, i.e., AdaBoost and Bagging algorithm, result in smaller misclassification errors even though the trees selected as a result of the Bagging or the Boosting in order to feed

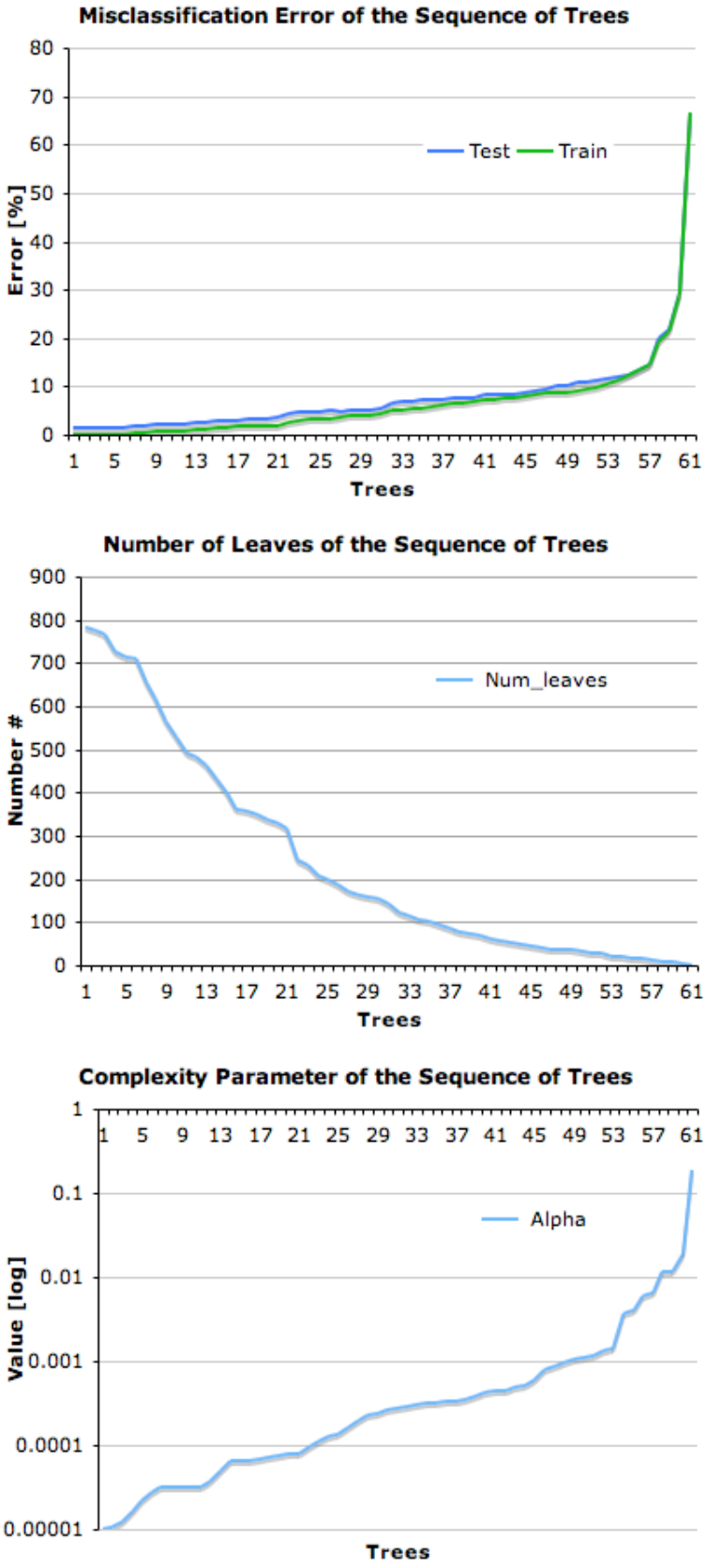

Figure 5-23: Minimal Cost-Complexity Pruning Statistics, Nursery Database, Multiple Classification

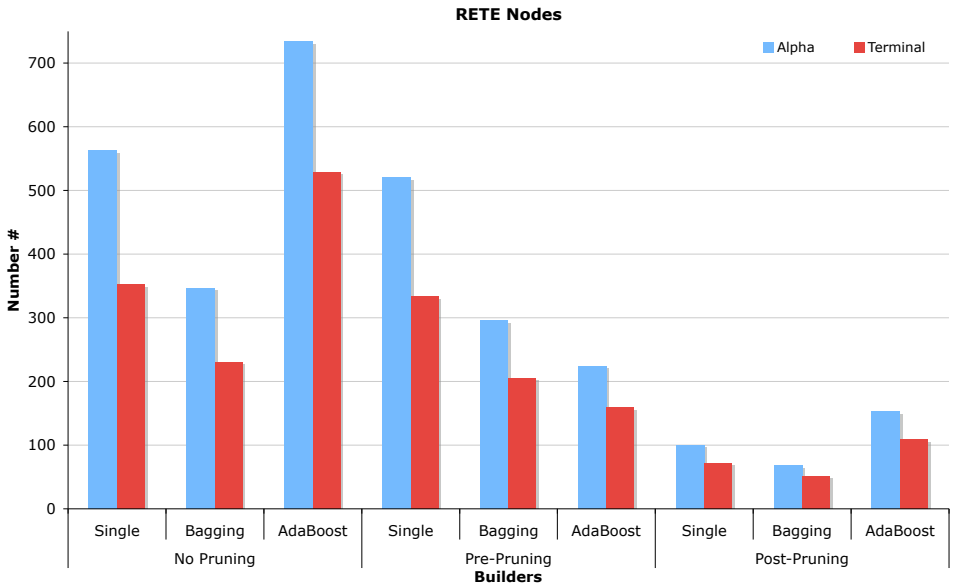

Figure 5-24: The Comparison of RETE Network Statistics between Builders using Poker Hands Database

to the RETE Network do not have classification errors as small as the classifier itself. Thus, the merging causes an increase in the classification error compared to the classifier. AdaBoost statistics are better than the any other builder since the test errors are lower than the Single Tree Builder.

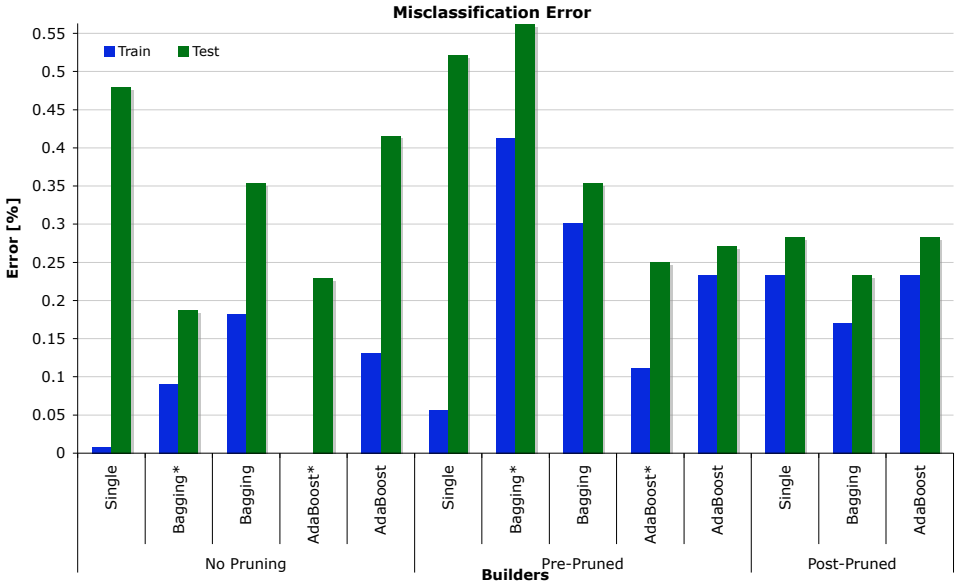

Figure 5-25: The Comparison of Misclassification Errors between Builders, using Pruned and Unpruned Trees

Table 5.17: The Comparison of Classification Results on Training and Test Set between Builders using Poker Hands Database

		Training					Testing				
Pruning Type	**Builder Type**	**incorrect N**	**incorrect N[%]**	**correct N**	**correct N[%]**	**total N**	**incorrect N**	**incorrect N[%]**	**correct N**	**correct N[%]**	**total N**
.	Single	2	0.008	25208	99.992	25210	23	0.479	4779	99.521	4802
.	Bagging*	23	0.091	25187	99.909	25210	9	0.187	4793	99.813	4802
No	Bagging	46	0.182	25164	99.818	25210	17	0.354	4785	99.646	4802
Pruning	AdaBoost*	0	0	25210	100	25210	11	0.229	4791	99.771	4802
.	AdaBoost	33	0.131	25177	99.869	25210	20	0.416	4782	99.584	4802
.	Single	14	0.056	25196	99.944	25210	25	0.521	4777	99.479	4802
.	Bagging*	104	0.413	25106	99.587	25210	27	0.562	4775	99.438	4802
Pre-	Bagging	76	0.301	25134	99.699	25210	17	0.354	4785	99.646	4802
Pruned	AdaBoost*	28	0.111	25182	99.889	25210	12	0.25	4790	99.75	4802
.	AdaBoost	59	0.234	25151	99.766	25210	13	0.271	4789	99.729	4802
.	Single	56	0.233	23953	99.767	24009	17	0.283	5986	99.717	6003
Post-	Bagging	41	0.171	23968	99.829	24009	14	0.233	5989	99.767	6003
Pruned	AdaBoost	56	0.233	23953	99.767	24009	17	0.283	5986	99.717	6003

Table 5.18: The Comparison of RETE Network Statistics between Builders using Poker Hands Database

Pruning	**Builder**	**ObjectType**	**Alpha**	**Join**	**Terminal**
.	Single	1	563	0	352
No	Bagging	1	347	0	230
Pruning	AdaBoost	1	734	0	529
.	Single	1	521	0	334
Pre-	Bagging	1	296	0	205
Pruning	AdaBoost	1	224	0	159
.	Single	1	100	0	72
Post-	Bagging	1	68	0	52
Pruning	AdaBoost	1	153	0	109

Table 5.19: The Pre-Pruner Statistics using Poker Hands Database

Builder	**EstimatedNodeSize**	**ImpurityDecrease**	**MaximumDepth**
Single	0	2	12
Bagging	19.5	0.2	3.5
AdaBoost	10.9	20.8	6.3

Chapter 6

Related Work

Pemer et al. describes and compares many discretization methods which explores different types of impurity measures [16]. Quinlan talks about the weakness of C4.5 in domains with continuous attributes and as a solution he presents the Minimum descriptive length – MDL – metrics [20]. It is notable that the WEKA project [23] uses the Fayyad and Irani's MDL method to discretize the numerical domains. Fayyad and Irani used information gain approach while evaluating the effectiveness of the discretization [7].

Knobbe gives the outlines of Multi-Relational Data Mining, a paradigm that is concerned with structured data in relational form [12]. Knobbe explains structured features by two classes: existential features, which express the presence of specific substructures, and aggregate functions, which express global properties of groups of parts. His algorithm is based on selection graphs can be used to select individuals purely on the basis of existential features. The main advantage of these graphs are one-to-one convertibility to the query languages so that it can be used on databases. Leiva implemented a Multi-Relational Decision Tree Learning (MRDTL) Algorithm for induction of decision trees from relational databases consisting of multiple tables and associations [13]. Leiva uses the algorithms and the structures designed by Knobbe.

There are different approaches for boosting. Freund introduces the idea of boosting C4.5 algorithm [10]. Hao generalizes the AdaBoost algorithm to Multi-class AdaBoost, AdaBoostK [11]. Zhu et al. proposes a new algorithm SAMME that naturally extends the original AdaBoost algorithm to the multi-class case without reducing it to multiple two-class problems [25]. Quinlan examines the application of bagging and boosting to C4.5 algorithm in his paper [19].

Chapter 7

Conclusions and Future Work

We showed that there can be many trees with same classification results but different sizes. The decision tree constructed using the learning algorithm tries to minimize the size of the RETE Network which is the most important criteria for Drools.

As we observed from the results the Multiple-Decision Tree Builders, i.e., Bagging and Boosting, improves the misclassification results. However, whenever we merge the multiple trees into one in order to provide the rules to the Drools the classifier looses accuracy. The power of the Multiple-Decision Tree Builders come from their voting schema. All trees not just one vote to decide on the classification results.

Using structured data we create structured rules for the RETE network. The number of AlphaNodes decreases in the structured rules whereas the number of JoinNodes increases due the structured nature.

We next consider several possibilities for future work:

1. *Concluding Instances as Unclassified*: The current version of the learning algorithm always classifies the instances to a category of the target attribute even if there is no attribute left in order to continue. We select the category which gets the maximum number of votes from the current set of instances as the target category. However, this causes many misclassification error since even 30 % of the instances vote in one category and all the other categories could get a number of votes more than 30 % of the total. One can say that the decision tree should conclude these instances on that branch of the tree as unclassified instead of classifying and causing a lot of misclassification error. Moreover, one can ask how the decision tree should classify the instances if there are more that one target categories which have the number of votes. An other situation that is similar to the 'no attribute left' situation is when the tree stops branching because of the stopping criteria condition is met. The current version of the tree learning algorithm selects the category which gets the majority of the votes. Whenever all elements are not classified to the same value and the branching can not continue the learning algorithm can decide for the node and the instances matching to it as unclassified if the majority is not big enough, i.e., 80 %.

   The Decision Tree Factory can ask for feedback for further classification from the user. For this purpose we can build a feedback mechanism into the tree factory that would tell if something did not get classified. Thus, this will decrease the misclassification error and increase the quality of the rules.

2. *Unknown Attribute Values*: The Object can be created with some of the attributes values missing. The Decision Tree Builder should be capable of tackling these kinds of situation. Even if there are instances with missing values the learner can continue using an estimation to select the attribute to branch.

3. *Unknown Target*: It is possible that the target to classify the instances is unknown. In this case we need to use unsupervised learning since the instances are not labeled by a supervisor. There are at least two situation possible. Either the user does not know which of the attributes is the target of the classification or she does not know how to categorize the instances and how many classes of the facts exist so the target attribute is not even one of the objects' attributes. On the other hand, the target does not have to be one of the attributes given. We can use the C4.5 algorithm to select the best of the given attributes as the target attribute. However, that is not very efficient because the idea is constructing a tree for each possible target attribute that would result in N trees if there are N attributes and then selecting the best tree with the maximum information. Moreover, we can use unsupervised learning algorithms to cluster the objects and to create a new target attribute. There are many clustering algorithms that we can use, i.e., k-means clustering

4. *Re-training Decision Trees* The Re-Training algorithm can be improved using an online version which ignore some part of the past data or keeps a statistics on it instead of saving the whole data.

5. *Grouping Discrete Attributes.* The literal attributes (Section 2) can have many possible values should be divided into subgroups in order to increase the performance.

6. *Multi-Relational Decision Trees.* Current version of the algorithm can not tackle an attribute which is collection of the other object or if the attribute is self-referencing, i.e., object is the attribute of herself. We should improve the Multi-Relational characteristics of the algorithm in the future.

Bibliography

[1] A. Asuncion and D. Newman. UCI machine learning repository, 2007. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[2] M. Bohanec and V. Rajkovic. Expert system for decision making. *Sistemica*, 1(1):145–157, 1990.

[3] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and regression trees*. Wadsworth, 1984.

[5] R. Cattral, F. Oppacher, and D. Deugo. Evolutionary data mining with automatic rule generalization, 2002.

[6] R. Doorenbos. *Production matching for large learning systems*. PhD thesis, Carnegie Mellon University, Jan 1995.

[7] U. Fayyad and K. Irani. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, Jan 1992.

[8] C. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *IEEE Computer Society Reprint Collection*, Jan 1991.

[9] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[10] Y. Freund and R. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, 1999.

[11] W. Hao and J. Luo. Generalized multiclass adaboost and its applications to multimedia classification. *CVPRW'06: Proceedings of the 2006 Conference on Computer Vision and Pattern Recognition Workshop*, 2006.

[12] A. J. Knobbe. *Multi-Relational Data Mining*. PhD thesis, Proefschrift Universiteit Utrecht, Jan 2004.

[13] H. Leiva. MRDTL: A multi-relational decision tree learning algorithm. Master's thesis, Iowa State University, 2002.

[14] D. Michie, D. J. Spiegelhalter, C. C. Taylor, and J. Campbell, editors. *Machine learning, neural and statistical classification*. Ellis Horwood, Upper Saddle River, NJ, USA, 1994.

[15] R. Mohan. Norvig and Russell's Artificial Intelligence - a modern approach online code repository, 2007. http://code.google.com/p/aima-java.

[16] P. Perner and S. Trautzsch. Multi-interval discretization methods for decision tree learning. *Advances in Pattern Recognition, Joint IAPR International Workshops SSPR '98 and SPR '98*, pages 475–482, Jan 1998.

[17] M. Proctor. JBoss Rules, Drools, December 2007. http://www.jboss.org/drools.

[18] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Jan 1993.

[19] J. R. Quinlan. Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 725–730. AAAI Press, 1996.

[20] J. R. Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

[21] R. A. R. Kohavi, F. Provost. Glossary of terms. *Mach. Learn.*, 30(2-3):271–274, 1998.

[22] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, San Francisco, CA, USA, Jan 2003.

[23] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Jan 2005.

[24] Y. Yohannes and P. Webb. *Classification and regression trees: a user manual for identifying indicators of vulnerability to famine and chronic food insecurity*, 1998.

[25] J. Zhu, S. Rosset, H. Zou, and T. Hastie. Multi-class adaboost. Technical report, Dept. Statistics University of Michigan, 2005.