Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

An Unified Overview of Six Supervised Symbolic Machine Learning Inducers

José Augusto Baranauskas
Maria Carolina Monard/ILTC

Nº 103

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
Fevereiro/2000

An Unified Overview of Six Supervised Symbolic Machine Learning Inducers*

José Augusto Baranauskas
Maria Carolina Monard/ILTC

University of São Paulo
Institute of Mathematics and Computer Sciences
Department of Computer Science and Statistics
Laboratory of Computational Intelligence
P.O. Box 668, 13560-970 - São Carlos, SP, Brazil
e-mail: {jaugusto, mcmonard}@icmc.sc.usp.br

Abstract

Gathering enough background and information related to specific Machine Learning inducers is not an easy task. In general, individual inducer documentation is spread out over several papers, books and internet sites. Even in the case of easy access to appropriate documentation sources, most of the time there is little similarity in the terminology adopted to describe each inducer.

This work presents an unified overview of supervised classification algorithms that learn concepts represented by propositional descriptions, such as decision trees and *if-then* rules.

After briefly reviewing some concepts about induction of decision tree as well as rule induction, the inducers $\mathcal{C}4.5$ and $\mathcal{OC}1$ for induction of decision trees, $\mathcal{C}4.5$rules, $\mathcal{CN}2$, and $\mathcal{R}ipper$ for rule induction, as well as $\mathcal{C}5.0$ for decision trees and rule induction, are treated in some detail within a common framework. Aiming to help the reader with the practical use of these inducers, the necessary input and the output produced for each of them while learning from a set of instances is also shown.

Keywords: Supervised Learning, Classification, Inducer.

February 2000

*Work partially supported by National Research Councils — CAPES and FINEP, Brazil.

This document was produced with the LaTeX typeset system and the BibTeX reference management system with help of the BibView tool (Prati et al., 1999). As with all reviewing work, it almost certainly contains errors and has plenty of room for improvements. Please report any error, typos, inconsistences, omissions and suggestions for improvements to jaugusto@icmc.sc.usp.br.

This document and possible updates can be found at the ICMC site:

ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_103.ps.zip

© Copyright 2000 by José Augusto Baranauskas & Maria Carolina Monard
All Rights Reserved

Contents

1 Introduction 1

2 Top Down Induction of Decision Trees 1
2.1 Building a Decision Tree . . . 2
2.2 Choosing the Best Feature to Split . . . 3
2.3 Tree Pruning . . . 3
2.4 Classifying New Instances . . . 3
2.5 Summary . . . 4

3 Rule Induction 4
3.1 Ordered Rule Induction . . . 4
3.2 Unordered Rule Induction . . . 5
3.3 Summary . . . 5

4 C4.5 5
4.1 Inducing Trees . . . 6
4.2 Choosing the Best Feature to Split . . . 6
4.3 Handling Continuous Values . . . 6
4.4 Handling Unknown Values . . . 7
4.5 Handling Noisy Instances . . . 7
4.6 Classifying New Instances . . . 7
4.7 Program Names . . . 8
4.8 Files . . . 8
4.9 C4.5 Options . . . 9
4.10 Example . . . 10
4.10.1 Input Data . . . 11
4.10.2 The Classifier . . . 12
4.10.3 Consulting the Classifier . . . 12
4.11 URL . . . 12
4.12 Summary . . . 15

5 C4.5rules 15
5.1 Inducing Rules from Tree . . . 15
5.2 Classifying New Instances . . . 16
5.3 C4.5rules Options . . . 16
5.4 Example . . . 16
5.4.1 Input Data . . . 17
5.4.2 The Classifier . . . 17
5.4.3 Consulting the Classifier . . . 17
5.5 URL . . . 17
5.6 Summary . . . 17

6 C5.0 20
6.1 Program Names . . . 20
6.2 Files . . . 20
6.3 C5.0 Options . . . 22
6.4 Example . . . 23
6.4.1 Input Data . . . 23

6.4.2 The Tree Classifier . . . 23
6.4.3 The Rule Classifier . . . 24
6.4.4 Consulting the Classifier . . . 25
6.5 URL . . . 25
6.6 Summary . . . 25

7 $\mathcal{OC}$1 28
7.1 Inducing Trees . . . 28
7.2 Choosing the Best Feature to Split . . . 29
7.3 Handling Unknown Values . . . 29
7.4 Handling Noisy Instances . . . 29
7.5 Classifying New Instances . . . 29
7.6 Files . . . 30
7.7 $\mathcal{OC}$1 Options . . . 30
7.8 Example . . . 33
7.8.1 Input Data . . . 33
7.8.2 The Classifier . . . 33
7.8.3 Consulting the Classifier . . . 35
7.9 URL . . . 35
7.10 Summary . . . 35

8 $\mathcal{CN}$2 36
8.1 Ordered $\mathcal{CN}$2 . . . 36
8.1.1 Inducing Rules . . . 36
8.1.2 Handling Continuous Values . . . 36
8.1.3 Handling Unknown Values . . . 37
8.1.4 Handling Noisy Instances . . . 37
8.1.5 Classifying New Instances . . . 37
8.2 Unordered $\mathcal{CN}$2 . . . 38
8.2.1 Inducing Rules . . . 38
8.2.2 Classifying New Instances . . . 38
8.2.3 Handling Unknown Values . . . 38
8.2.4 Handling Noisy Instances . . . 39
8.3 Program Name . . . 39
8.4 Files . . . 39
8.5 $\mathcal{CN}$2 Options . . . 41
8.6 Batch Mode . . . 42
8.7 Example . . . 42
8.7.1 Input Data . . . 42
8.7.2 The Ordered Rules Classifier . . . 42
8.7.3 The Unordered Rules Classifier . . . 43
8.7.4 Consulting the Classifier . . . 44
8.8 URL . . . 44
8.9 Summary . . . 49

9 $\mathcal{R}ipper$ 49
9.1 Inducing Rules . . . 50
9.2 Handling Continuous Values . . . 50
9.3 Handling Unknown Values . . . 50
9.4 Handling Noisy Instances . . . 51

9.5 Classifying New Instances . . . 51
9.6 Program Names . . . 51
9.7 Files . . . 53
9.8 *Ripper* Options . . . 55
9.9 Example . . . 57
9.9.1 Input Data . . . 57
9.9.2 The Classifier . . . 57
9.9.3 Consulting the Classifier . . . 59
9.10 URL . . . 60
9.11 Summary . . . 60

10 Inducers Summary **61**

11 Concluding Remarks **61**

References **62**

List of Figures

1 A simple decision tree for diagnosing a patient . . . 2
2 A larger tree is first grown that overfits the data and then pruned back to a smaller (simpler) tree . . . 3
3 The $\mathcal{C}4.5$ `voyage.names` file . . . 11
4 The $\mathcal{C}4.5$ `voyage.data` file . . . 12
5 The voyage $\mathcal{C}4.5$ classifier . . . 13
6 Consulting the voyage $\mathcal{C}4.5$ classifier . . . 14
7 The voyage $\mathcal{C}4.5$rules classifier . . . 18
8 Consulting the voyage $\mathcal{C}4.5$rules classifier . . . 19
9 The voyage $\mathcal{C}5.0$ tree classifier . . . 24
10 The voyage $\mathcal{C}5.0$ rule classifier . . . 26
11 Consulting the $\mathcal{C}5.0$ voyage decision tree classifier . . . 27
12 Consulting the $\mathcal{C}5.0$ voyage rule classifier . . . 27
13 The $\mathcal{OC}1$ `voyage.data` file . . . 34
14 The voyage $\mathcal{OC}1$ classifier . . . 34
15 The voyage $\mathcal{OC}1$ unpruned classifier . . . 34
16 Consulting the $\mathcal{OC}1$ voyage classifier . . . 35
17 The $\mathcal{CN}2$ `voyage.att` file . . . 43
18 The $\mathcal{CN}2$ `voyage.exs` file . . . 43
19 The `voyage.cn2.ordered-commands` batch file . . . 44
20 The voyage $\mathcal{CN}2$ ordered rule classifier output . . . 45
21 The `voyage.cn2.unordered-commands` batch file . . . 46
22 The voyage $\mathcal{CN}2$ unordered classifier . . . 46
23 Consulting the voyage $\mathcal{CN}2$ ordered classifier . . . 47
24 Consulting the voyage $\mathcal{CN}2$ unordered classifier . . . 48
25 The *Ripper* `voyage.names` file . . . 58
26 The *Ripper* `voyage.data` file . . . 58
27 The voyage *Ripper* classifier . . . 59
28 The voyage grammar . . . 59
29 Consulting the *Ripper* voyage rule classifier . . . 60

List of Tables

1 The voyage data 11
2 Inducers Summary 61

List of Algorithms

1 $\mathcal{CN}2$ Ordered rule induction 37
2 $\mathcal{CN}2$ Unordered rule induction 39
3 $\mathcal{CN}2$ Find best conjunction 40
4 IREP 50
5 $\mathcal{R}ipper$ 51
6 $\mathcal{R}ipper$ Optimize Rule Set 51
7 $\mathcal{R}ipper$ Build Rule Set 52

1 Introduction

> We're all human and we all goof. Do things that may be wrong, but do something.
>
> —*Newt Gingrich, US Congressman and House Speaker, 1989*

The main goal of this work consists in providing some background about Machine Learning — ML — inducers by describing algorithms that learn concepts represented by propositional descriptions, specifically decision trees and rules. We have tried to unify the notation used in both cases, aiming to yield a better understanding to the reader. We discuss six of this sort of inducers commonly used by the ML community: $\mathcal{C}4.5$ and $\mathcal{OC}1$ for decision trees and $\mathcal{C}4.5$rules, $\mathcal{CN}2$ and $\mathcal{R}ipper$ for rules as well as $\mathcal{C}5.0$ for decision trees and rules.

This work was motivated mainly by a lack of an unified view in the literature for describing widely used inducers. Furthermore, getting background knowledge on what each inducer does, how they work and how to use each of them is not a simple task. Besides that, existing documentation about those inducers does not necessarily follow a standard form, which makes more difficult the understanding for beginners in the area of ML. However, this work does not intend to substitute any of the original papers. It should be observed that those papers are very relevant and anyone interested in gathering a deeper ML knowledge and understanding should read the original papers and books listed in the references.

This work is organized as follows. Section 2 briefly describes induction of decision trees and Section 3 shows the rule induction process. Sections 4 to 9 present characteristics of each inducer treated in this work ($\mathcal{C}4.5$, $\mathcal{C}4.5$rules, $\mathcal{C}5.0$, $\mathcal{OC}1$, $\mathcal{CN}2$ and $\mathcal{R}ipper$) as well as a running example using each of them. A brief summary of inducers characteristics is presented in Section 10 and, finally, concluding remarks are given in Section 11.

2 Top Down Induction of Decision Trees

> Somewhere, something incredible is waiting to be known.
>
> —*Carl Sagan*

We start this section explaining basic concepts about inductive learning when acquired knowledge is represented as a decision tree. This sort of algorithms are members of the Top Down Induction of Decision Trees — TDIDT — family.

A Decision Tree — DT — is a recursive data structure defined as:

- a *leaf node* that indicates a class label, or
- a *decision node* that contains a test on a feature value. For each outcome of the test there are one branch and one subtree. Each subtree has the same structure as the tree.

Figure 1 shows an illustrative example of a decision tree for diagnosing a patient. In this figure, each ellipse is a test in one feature from some patient data. Each box is a class label *i.e.*, the diagnosis. To diagnose (classify) a patient we start at the root just following each test down the tree until a leaf is reached.

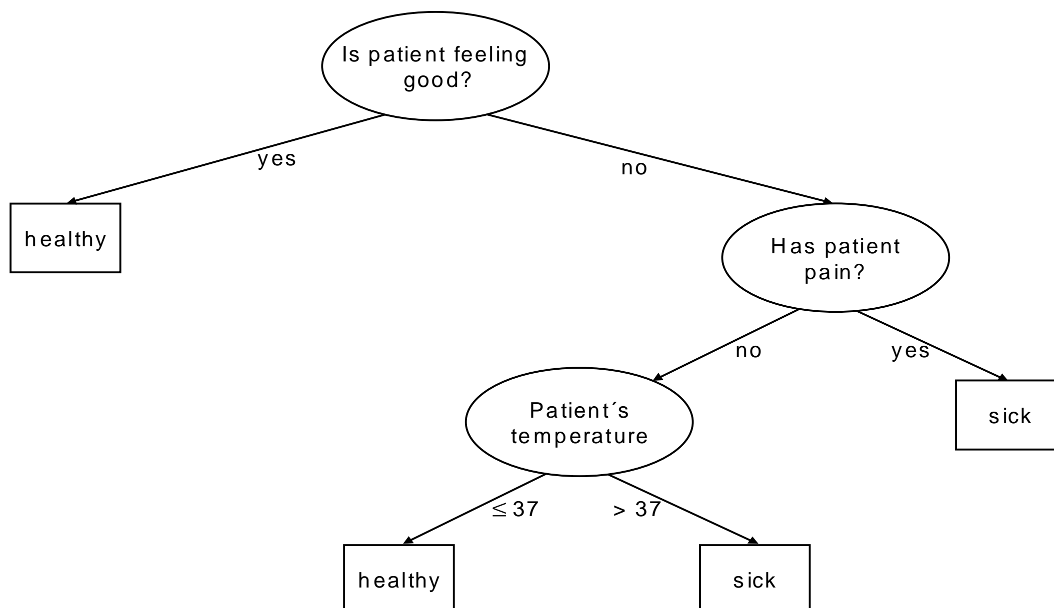

Figure 1: A simple decision tree for diagnosing a patient

2.1 Building a Decision Tree

The method for constructing a decision tree from a set T of training instances is surprisingly simple. Let the classes be denoted by $\{C_1, C_2, \ldots, C_k\}$ then the following steps should be followed:

1. T contains one or more instances, all belonging to a single class C_j. In this case, the DT for T is a leaf identifying the class C_j;
2. T contains no instances. Again, in this situation the DT is a leaf but the class associated with the leaf must be determined from information other than T. For example, the most frequent class at the parent of this node could be used;
3. T contains instances that belong to several classes. In this case the idea is to refine T into subsets of instances that are (or seem to be) single-class sets of instances. Normally, a test is chosen, based on a single feature, that has one or more mutually exclusive outcomes (in fact, each inducer has its own way to choose the feature to test). Let us denote these outcomes as $\{O_1, O_2, ..., O_r\}$. T is then partitioned into subsets T_1, T_2, ..., T_r, where T_i contains all cases in T that have outcome O_i for the chosen test. The DT for T consists of a internal decision node identified by the chosen test and one branch for each possible outcome;
4. Steps 1., 2. and 3. are applied recursively to each subset of training instances so that the i-th branch leads to the DT constructed from the subset T_i of training instances;
5. After building the DT, pruning may take place (see Section 2.3).

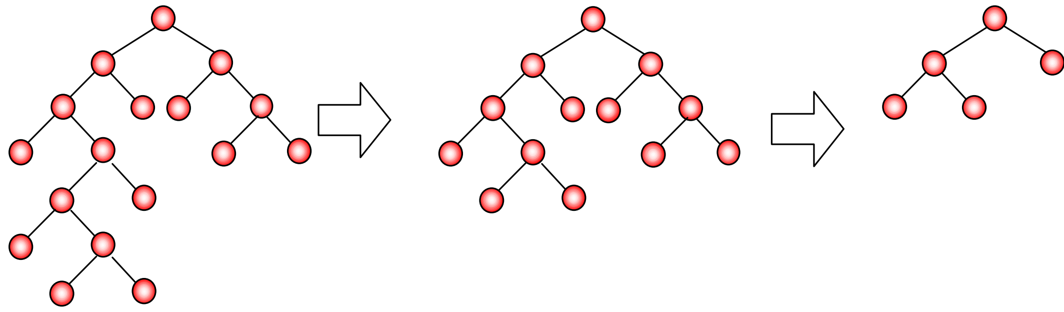

Figure 2: A larger tree is first grown that overfits the data and then pruned back to a smaller (simpler) tree

2.2 Choosing the Best Feature to Split

The key to the success of a DT learning algorithm depends on the criterion used to select the feature for splitting. Some possibilities for selecting this feature are *random* which selects any feature at random; *least–values* which chooses the feature with the smallest number of possible values; *most–values* which chooses the feature with the largest number of possible values; *max-gain* which chooses the feature that has the largest expected information gain; *gini* (Breiman et al., 1984); *gain ratio* (Quinlan, 1988).

2.3 Tree Pruning

After building the DT, it is possible that the generated classifier to be very specific for the training data. In this case, we say that the classifier *overfits* the training data too well. In order to try not overfitting the data, some inducers *prune* the DT after inducing it. This process, shown in Figure 2, reduces the number of internal test nodes thus reducing the tree complexity while giving a better performance than the original tree. In general, a DT inducer splits by itself the dataset into a training set (which is used for building the DT) and a prune set which is actually used for pruning.

2.4 Classifying New Instances

A DT can be used to classify new instances by starting at the root of the tree and going through each decision node until a leaf is found. When a leaf is encountered, the class label of the new instance is predicted as that one recorded at the leaf.

As there is only one path from the root to the leaf of every tree, each path from the root to each leaf can be considered as a rule. In this case, given an instance it can be classified only by one rule, or in other words, each instance classified by one rule can not be classified by another rule. It is easy to see that all rules extracted from a decision tree are always disjunct.

2.5 Summary

> Only the curious will learn and only the resolute overcome the obstacles to learning. The quest quotient has always excited me more than the intelligence quotient.
>
> —*Eugene S. Wilson*

Induction of decision trees is one of the most widely used learning methods in practice. It can outperform human experts in many domains. Some of it strengths is due to be a fast method for learning concepts; it is simple to implement; it ca handle noisy data, etc. However larger trees are generally difficult to read, even after tree pruning.

3 Rule Induction

> I think, at a child's birth, if a mother could ask a fairy godmother to endow it with the most useful gift, that gift would be curiosity.
>
> —*Eleanor Roosevelt*

As described in the previous section, decision tree induction divides recursively the data into smaller subsets, trying to separate each class from others. Rule induction, on the other hand, induces rules directly. In this process each rule covers a subset of instances that belongs to one specific class.

A rule assumes the form

$$\textbf{if} <\text{complex}> \textbf{then} <\text{class} = C_i>$$

where C_i belongs to the set of possible k class values and a <complex> is a disjunction of conjunctions of feature tests in the form:

$$X_i \; op \; Value$$

where X_i is a feature, *op* is an operator in the set $\{=, \neq, <, \leq, >, \geq\}$ and *Value* is a valid feature X_i constant value.

In general, the Disjunctive Normal Form — DNF — is the most common representation used by many inducers. Thus, a rule is a disjunction of conjunctive conditions. Basically, there are two sort of rule induction: *ordered* and *unordered* which are explained in the following sections.

3.1 Ordered Rule Induction

The induction of ordered rules works in an iterative way, each iteration searching for a <complex> which covers a large number of instances of a single class C_i and few of other classes $C_j, j \neq i$.

Having found a good <complex>, those covered instances that belong to the class C_i being learned (as well as, eventually, other few instances having class C_j, $j \neq i$ also covered by the same <complex>) are removed from the training set and the rule "if <complex> then class $= C_i$" is added to the end of the rule list. This process iterates until no more complexes can be found.

3.2 Unordered Rule Induction

The main modification to the ordered algorithm is to iterate for each class in turn, removing only covered instances of that class when a rule has been found. Thus, unlike for ordered rules, the instances from classes C_j, $j \neq i$ incorrectly covered by the current <complex> should remain because now each rule must independently stand against all incorrectly covered instances. Covered instances having the correct class C_i being learned must be removed to stop repeatedly finding the same rule.

3.3 Summary

> Personally, I'm always ready to learn, although I do not always like being taught.
>
> —*Winston Churchill*

The strengths of rule induction is comprehensibility and low storage requirements. However, the process of inducing rules is a slower process than inducing decision trees and there are often many variables to tune.

In the following sections several Machine Learning algorithms that induce decision trees, as well as ordered and unordered rules, are described in some detail. Commands, options and input files to run each algorithm followed by a running example of each algorithm are also described.

4 $\mathcal{C}4.5$

> Experimentation with variants of current learning algorithms, specially when it is directed towards probing their weakness, should lead to a better understanding of how methods relate to families of tasks. This is essential if we are to design a new generation of more robust and effective learning systems.
>
> —*John Ross Quinlan, (Quinlan, 1988)*

$\mathcal{C}4.5$ is a member of the TDIDT family *i.e.*, $\mathcal{C}4.5$ creates decision trees to represent classification rules (Quinlan, 1988). As stated earlier, a node in a decision tree represents a test on a particular feature.

Suppose an object is described by its color, size and shape in the following way:

- colors may have values "red", "green" or "blue";
- size may be "small", "medium" or "large";
- shape may be "circle" or "square".

Thus, if the root node of the tree contains the color feature then it may have three branches, one for each color value. As explained in Section 2.2 (page 3) if we wish to classify a new unlabeled instance and its color is red, we have to descend through the red branch. As leaf nodes are labeled with class names when a leaf node is reached, the new instance will be classified with the name of the leaf node.

4.1 Inducing Trees

Building a decision tree in $\mathcal{C}4.5$ proceeds as follows (Quinlan, 1986). All examples are divided in two disjoint subsets: the training set and the prune set. Using the training set, a feature is chosen to split it according to feature values. Indeed, if color is chosen then all red objects descend the red branch, all green objects descend the green branch and so on. After that, the original training set is *partitioned* by color into smaller subsets. For each subset, another feature is chosen for splitting. This continues as long as each subset contains instances belonging to different classes. Once a uniform subset — *i.e.*, all instances in that subset belongs to the same class — has been obtained, a leaf node is created and labeled with the same name of the respective class.

In $\mathcal{C}4.5$ each leaf node assumes the form:

$$<C_i> \ (N) \text{ or}$$
$$<C_i> \ (N/E)$$

where C_i is one class label and N is the sum of the fractional instances that reach that leaf; E is the number of instances that belong to classes other than class C_i for unpruned trees. For pruned trees, E is just the number of predicted errors if a set of N unseen instances were classified by the tree.

4.2 Choosing the Best Feature to Split

The key to the success of a decision tree learning algorithm depends on the criterion used to select the feature for splitting. If a feature is a strong indicator of an instance's class value, it should appear as early in the tree as possible — near the root.

Most decision tree learning algorithms use a heuristic for estimating the best feature. $\mathcal{C}4.5$ uses a modified version of the entropy measure from information theory. For our purposes, it is sufficient to state that this measure yields a number between 0 and 1, where 0 indicates an uniform set (only one class value is present for all instances of that feature) and 1 indicates a set where there is equal likelihood of all class values being present. The splitting criterion searchs for minimizing the entropy. Specifically, $\mathcal{C}4.5$ uses the gain ration criterion to choose the best feature to split.

4.3 Handling Continuous Values

$\mathcal{C}4.5$ deals with continuous features by dividing the range of those features into discrete sub-ranges. Tests on such features assume the form:

$$X_i \ op \ Value$$

where X_i is a feature, *op* is an operator in the set $\{<, \leq, >, \geq\}$ and *Value* is a valid feature X_i constant value. *Value* is found by sorting all values for feature X_i for a dataset with n instances. Let us denote the sorted values for feature X_i as $\{x_1, x_2, \ldots, x_n\}$, assuming the worst case when all instances have different X_i values. Consequently, any threshold value lying in the range $\{x_j, x_{j+1}\}$ will divide instances into those whose value of feature X_i lies between

$$\{x_1, x_2, \ldots, x_j\} \text{ and } \{x_{j+1}, x_{j+2}, \ldots, x_n\}$$

Observe that there are only $(n-1)$ possible splits on X_i. Since values $\{x_1, x_2, \ldots, x_n\}$ are sorted, it is easy to update the class distribution to the left and to the right of the threshold.

It is usual to choose the midpoint of each interval $\{x_j, x_{j+1}\}$ given by

$$\frac{x_j + x_{j+1}}{2}$$

as the representative threshold. However, $\mathcal{C}4.5$ chooses the largest value of X_i in the entire training set that does not exceed the midpoint of each interval, rather than the midpoint itself, ensuring that all values appearing in the tree actually occur in the data.

Afterwards, the values of gain ratio from division based on the $(n-1)$ possible thresholds are calculated and the division that gives a greater gain ratio becomes the root of the decision tree (or the current subtree).

4.4 Handling Unknown Values

The treatment of missing feature values — represented as '?' — in $\mathcal{C}4.5$ follows information theory as well. Let T be the training set and $t(X_i)$ a test based on some feature X_i. Suppose that only a fraction F of feature X_i's value is known. In this case $\mathcal{C}4.5$ takes this into account when computing the information gain: it considers only the fraction F of known values.

4.5 Handling Noisy Instances

A further refinement is required to handle noisy data. Real datasets often contain instances that are misclassified or which have incorrect feature values or even are inconsistent. Suppose that — in a two class problem — the decision tree building has constructed a node with 999 instances from class 1 and only one instance from class 2. According to the general TDIDT algorithm, a further split would be required to separate the 999 class 1 instances from just one instance in class 2.

However, the one exception may be mislabeled or noise, causing an unnecessary split. Decision tree learning algorithms have a variety of methods for *pruning* unwanted subtrees. $\mathcal{C}4.5$ grows a complete tree, including nodes created as result of noise. Following initial tree building, the algorithm proceeds to select *suspect* subtrees and *prune* them, testing the new tree on a dataset which is separated from the initial training set *i.e.*, the prune set. Pruning continues as long as the pruned tree yields more accurate classifications on the test data (Quinlan, 1987b).

4.6 Classifying New Instances

When a new instance should be classified, beginning from the root of the induced tree, $\mathcal{C}4.5$ tests-and-branches each node with the respective feature until it reaches one leaf. The class prediction of this instance is then assigned as the class of that leaf.

4.7 Program Names

c4.5 This command invokes decision tree building (see Section 4.9 for options of this command).

c4.5rules This command invokes rule induction. It should only be used after running the decision tree c4.5, since it reads the `.unpruned` file created by this program.

consult After inducing decision trees with c4.5, this program can be used to consult the generated classifier. This program requests values of features from the user. For discrete features, valid user answers are:

- "?" indicating unknown feature value;
- a single value;
- a set of possible values in the form $x_1 : p_1, x_2 : p_2, ..., x_r : p_r$ where x_i are possible values and p_i are the corresponding value probabilities.

For real valued features, valid user answers are:

- "?" indicating unknown feature value;
- a single number;
- a range consisting of two numbers separated by a hyphen. In this case the feature value is assumed as being uniformly distributed in the range.

Each user answer must be followed by a return. After requesting all needed data the classification for that particularly instance is showed.

consultr This program is used to consult rules induced by c4.5rules. Valid user answers are the same as for consult.

4.8 Files

All files read and written by $\mathcal{C}$4.5 assume the form *filestem.extension*, where *extension* characterizes the type of data involved:

`.names` This text file provides names for classes, features and enumerated possible feature values. Each line ending with a period. Blank lines, spaces and tabs can be used for clarity. Anything from the vertical bar (|) to the end of line is ignored and can be used for comments.

The first entry must be the class values, separated by commas and terminating with a period. There must be at least two class names, no matter their order.

The remaining entries consist of one line for each feature. A feature description begins with feature name (no comma, colon, vertical bar and blackslash are allowed in feature names) followed by a colon and then possible values the feature can take (ended with a period). The possible values are:

- ignore causing the feature to be completely ignored;
- continuous indicating integer or floating point values;

- discrete N where $N > 0$ is an integer specifying the feature can take no more than N discrete values;
- a list of names separated by commas, indicating the feature can take discrete values. For instance:

  color: red, green, blue.

  The difference between 'discrete N' and a list of names is that in the first case, $\mathcal{C}4.5$ cannot check feature values supplied in the `.names` file against the `.data` file as it is done whenever the list of values is used. Therefore, it is recommended to use a list of values instead of 'discrete N' whenever possible.

It should be observed that although the order of feature declaration is not relevant, it must follow the exact data position in the `.data` file. In other words, feature declaration defines the feature value position in the `.data` file.

`.data` Each line from this text file describes one instance, providing values for each feature. Each value must be separated by other using commas and optionally terminated by period. The feature values must appear exactly in the same order as they are declared in the `.names` file. Unknown values can be specified by a question mark. The order of instances is not relevant. This file is required for running c4.5 or c4.5rules.

Observe the class label must be the last column, despite it is declared first in the `.names` file.

`.test` This optional text file assumes the same form of the `.data` file. It is used to evaluate accuracy of the generated classifier. Normally, this file should not contain any instance present in the `.data` file.

`.unpruned` Output binary file containing unpruned trees generated by c4.5. This file is used by c4.5rules.

`.tree` Output binary file containing the final pruned tree. If more than one tree is generated via windowing, the best one is chosen. This file is used by consult.

`.rules` Output binary file containing the final rule set generated by c4.5rules. It is used by consultr.

4.9 $\mathcal{C}4.5$ Options

This sections describes $\mathcal{C}4.5$ command line options. For more details, consult (Quinlan, 1988). For running $\mathcal{C}4.5$, the command line assumes the form:

c4.5 [*options*]

Options can appear in any order and they can be:

- `-f` *filestem*
  Specify the input dataset. No extension should be given. If no *filestem* is given, the default value *DF* is assumed (therefore, *DF.names* and *DF.data* will be searched in the current path).

- `-u`
  Tell $\mathcal{C}4.5$ to use a test file (`.test`). The default is not to use a test file.

- `-s`
  Cause grouping on discrete values (in sets) for node tests. The default is no grouping, consequently each internal tree node branches separately for every possible feature value.

- `-m` *weight*
  Generally used for noisy data to avoid trees with little predictive power. It indicates that each test must have at least *weight* outcomes with a minimum number of instances (the sum of the weights of the instances for at least two of the subsets must attain some minimum). Higher values may be interesting for tasks with noisy data. The default value is 2.

- `-c` *CF*
  The *CF* value affects tree pruning. Small values cause more heavy pruning than large values. The default value is 25%.

- `-v` *level*
  Indicate the verbosity level to be used, from 0 to 5. Level 3 and above generates lots of amount of output. The default value is 0.

- `-t` *trees*
  Invoke windowing, in which trees are grown iteratively. It specifies the number of trees to be grown before one is selected as the best one. One of the following windowing options should be used or the program will grow a single tree in the standard way. The default value is 10.

- `-w` *size*
  Activate windowing and give the number of instances that can be included in the initial window. The default value is $\max\{0.2n, 2\sqrt{n}\}$, where n is the number of training instances.

- `-i` *increment*
  Activate windowing and give the maximum number of instances that can be added to the window at each iteration. However, at least half of the number of training instances misclassified by the current rule are added to the next window, no matter the value of this option. The default value is 20% of the initial window size.

- `-g`
  Change the choice of the best feature from gain ratio to the older gain criterion. The default value is the most recent gain ration criterion.

- `-p`
  Enable subsidiary cutpoints (range) for test on continuous features. The default value is to use hard thresholds (one value).

4.10 Example

This section shows a running example, adapted from (Quinlan, 1988), for better understanding how $\mathcal{C}4.5$ builds a DT. In order to clarify the difference among the various machine learning approaches considered in this work, the same example will be used in the remaining sections.

Let us suppose a dataset containing day-by-day measures from weather conditions where each instance is composed of the following features:

- outlook: assumes discrete values "sunny", "overcast" or "rain";
- temperature: a numeric value indicating the temperature in Celsius degrees;
- humidity: also a numeric value indicating the percentage of humidity;
- windy: assumes discrete values "yes" or "no" indicating if it is a windy day.

Instance No.	Outlook	Temperature	Humidity	Windy	Voyage?
T_1	sunny	25	72	yes	go
T_2	sunny	28	91	yes	dont_go
T_3	sunny	22	70	no	go
T_4	sunny	23	95	no	dont_go
T_5	sunny	30	85	no	dont_go
T_6	overcast	23	90	yes	go
T_7	overcast	29	78	no	go
T_8	overcast	19	65	yes	dont_go
T_9	overcast	26	75	no	go
T_{10}	overcast	20	87	yes	go
T_{11}	rain	22	95	no	go
T_{12}	rain	19	70	yes	dont_go
T_{13}	rain	23	80	yes	dont_go
T_{14}	rain	25	81	no	go
T_{15}	rain	21	80	no	go

Table 1: The voyage data

Furthermore, for each day (instance), someone has labeled each day-by-day measure as "go" if weather was nice enough for taking a trip to the farm or "dont_go" if this is not the case. The data could look just like the one shown in Table 1. Although this example has only two possible class labels, it is important to recall that a $\mathcal{C}4.5$ and other DT inducers, can handle any number k of classes $\{C_1, C_2, \ldots, C_k\}$.

4.10.1 Input Data

Figures 3 and 4 show the corresponding `voyage.names` and `voyage.data`, respectively. For this example, no `voyage.test` file is used.

```
go, dont_go.        | Classes

| Features
outlook:     sunny, overcast, rain.
temperature: continuous.
humidity:    continuous.
windy:       yes, no.
```

Figure 3: The $\mathcal{C}4.5$ `voyage.names` file

```
sunny,   25,72,yes,go
sunny,   28,91,yes,dont_go
sunny,   22,70,no, go
sunny,   23,95,no, dont_go
sunny,   30,85,no, dont_go
overcast,23,90,yes,go
overcast,29,78,no, go
overcast,19,65,yes,go
overcast,26,75,no, go
overcast,20,87,yes,dont_go
rain,    22,95,no, go
rain,    19,70,yes,dont_go
rain,    23,80,yes,dont_go
rain,    25,81,no, go
rain,    21,80,no, go
```

Figure 4: The $\mathcal{C}4.5$ `voyage.data` file

4.10.2 The Classifier

For running $\mathcal{C}4.5$ with default options, we just need to specify the *filestem* to be used:

```
c4.5 -f voyage
```

Figure 5 shows the output generated by $\mathcal{C}4.5$.

4.10.3 Consulting the Classifier

After inducing the decision tree, we would like to know, based on information gathered from today weather if would be nice to take a trip. Our data about today could look like (outlook, temperature, humidity, windy) = (sunny, 35, 70, no) and (?, 30, 80, yes). Note that these instances do not exist in the training set presented to the inducer. We can use:

```
consult -f voyage -v
```

to get the answer. The `-v` option is used just for getting more details about the consulting process but it is not really needed. The output of the consult session is shown in Figure 6.

4.11 URL

You can find $\mathcal{C}4.5$ in the URL:

http://www.cse.unsw.edu.au/∼quinlan/

```
C4.5 [release 8] decision tree generator    Thu Oct  7 00:37:00 1999
----------------------------------------

    Options:
    File stem <voyage>

Read 15 cases (4 attributes) from voyage.data

Decision Tree:

outlook = overcast: go (5.0/1.0)
outlook = sunny:
|   humidity <= 78 : go (2.0)
|   humidity > 78 : dont_go (3.0)
outlook = rain:
|   windy = yes: dont_go (2.0)
|   windy = no: go (3.0)


Tree saved


Evaluation on training data (15 items):

     Before Pruning           After Pruning
    ----------------   ---------------------------
    Size      Errors   Size      Errors   Estimate

       8    1( 6.7%)      8    1( 6.7%)    (43.3%)   <<
```

Figure 5: The voyage $\mathcal{C}4.5$ classifier

```
C4.5 [release 8] decision tree interpreter      Sun Oct 10 01:09:57 1999
------------------------------------------

outlook: sunny
        Weight 1: test att outlook (val sunny = 1)
humidity: 70
        Weight 1: test att humidity (branch weight=1)
        Class go weight 1 cases 2
class 0 weight 1.00
class 1 weight 0.00

Decision:
        go  CF = 1.00  [ 0.50 - 1.00 ]

Retry, new case or quit [r,n,q]: n

-------------------------------------------

outlook: ?
humidity: 80
        Weight 0.333333: test att humidity (branch weight=0)
        Class dont_go weight 0.333333 cases 3
        Class go weight 0.333333 cases 5
windy: yes
        Weight 0.333333: test att windy (val yes = 1)
        Class dont_go weight 0.333333 cases 2
class 0 weight 0.27
class 1 weight 0.73

Decision:
        dont_go  CF = 0.73  [ 0.38 - 0.82 ]

Retry, new case or quit [r,n,q]: q
```

Figure 6: Consulting the voyage $\mathcal{C}4.5$ classifier

4.12 Summary

> This is a very well-written book, and any reader not familiar with Quinlan's writings will quickly discover that Quinlan's style is refreshingly clear. The examples are especially helpful, as they provide concrete illustrations of many of the algorithmic details of $\mathcal{C}4.5$. In part because the writing is so good, it would have been nice to have a longer book in which he surveyed some of the substantial body of literature on decision trees, such as pruning methods, alternative goodness criteria, and experimental results.
>
> —*Steven L. Salzberg, Book Review of (Quinlan, 1988)*

$\mathcal{C}4.5$ is one of the most (if not the most) famous inducer for decision trees. In general, its code is robust enough for running in several platforms and its restrictions about number of instances and features are only limited by the operating system.

In many situations, $\mathcal{C}4.5$ has been used as a baseline inducer for comparison with new ones. In fact, its performance is good enough in most of, but not all, domains.

5 $\mathcal{C}4.5$rules

> Discovery consists of seeing what everybody has seen and thinking what nobody has thought.
>
> —*Albert von Szent-Gyoergyi*

$\mathcal{C}4.5$rules is a learning algorithm that creates decision rules extracted from $\mathcal{C}4.5$ decision tree (Quinlan, 1987a). Therefore, before running $\mathcal{C}4.5$rules it is necessary to run $\mathcal{C}4.5$.

The motivation for $\mathcal{C}4.5$rules is that large decision trees are difficult to read. This is because each node has a specific context established by outcomes of tests at parent nodes.

5.1 Inducing Rules from Tree

The process of extracting rules from a tree can be as simple as generating one rule for each leaf. $\mathcal{C}4.5$rules does a little more work than that: for each rule (extracted from the unpruned tree) it removes one-by-one its conditions and it evaluates each resulting rule using the *pessimistic error* (Quinlan, 1988). The best condition to be removed from resulting rule is the one with the lowest pessimistic error. After that, this process is repeated until it is not possible to delete a rule condition without increasing its pessimistic error. Then, for each class, all simplified rules are evaluated in order to remove rules that do not contribute to the accuracy of the whole rule set.

After inducing the whole rule set, all rules for a single class appear together trying to minimize false positive errors (Baranauskas and Monard, 2000). This means that there is no order in rules predicting the same class. However, intra-class rules must obey the original order given by $\mathcal{C}4.5$rules. In other words, rules predicting the same class can be considered as unordered while intra-class rules can be seen as ordered. Finally, there is a default rule which is only activated if none of the other rules is satisfied. The default rule defined by $\mathcal{C}4.5$rules assigns the class which contains most training instances not covered by any induced rule.

In $\mathcal{C}4.5$rules each rule assumes the form:

<complex> –> <class C_i> [$A\%$]

which can be read as '*if* <complex> *then* <class = C_i> with accuracy of $A\%$'.

5.2 Classifying New Instances

When a new instance should be classified using the induced rules, $\mathcal{C}4.5$rules tries each rule in order until one is found whose conditions are satisfied by the new instance being classified. The resulting class prediction of this rule is then assigned as the class of that instance. If no rule is satisfied then the class of the default rule is assigned to the new instance.

Whenever the user decides to classify manually a new instance, it is very important to remember that the order of rules in each class should be respected.

5.3 $\mathcal{C}4.5$rules Options

This section describes $\mathcal{C}4.5$rules command line options. For running $\mathcal{C}4.5$rules, the command line assumes the form:

c4.5rules [*options*]

Options can appear in any order. The first four options have the same meaning as for $\mathcal{C}4.5$ — see Section 4.9, page 9:

- `-f` *filestem*
- `-u`
- `-v` *level*
- `-c` *CF*
  The *CF* value is used to prune rules instead of trees.
- `-F` *confidence*
  The <complex> of a rule is checked using Fisher's exact test. Each condition must be significant at the specified *confidence* level. In general, this produces shorter <complex> than the standard pruning method *i.e.*, generalizing tasks with little data.
- `-r` *redundancy*
  A *redundancy* value of 3 means there are 3 times more features than are likely to be useful for learning concepts. Consequently, this option allows the user to tell the inducer that there are many redundant features in the training set. It should be used only if the user knows very well the data. The default value is 1.0.

5.4 Example

In what follows the voyage data (Section 4.10, page 10) will be used to show a simple $\mathcal{C}4.5$rules execution.

5.4.1 Input Data

The input data for $\mathcal{C}4.5$rules is the same used for $\mathcal{C}4.5$ — see Figures 3 and 4, page 11, respectively.

5.4.2 The Classifier

For running $\mathcal{C}4.5$rules with default options, we just need to specify the *filestem* to be used:

```
c4.5rules -f voyage
```

Figure 7 shows the output generated by $\mathcal{C}4.5$rules. In the final rule set the predicted accuracy of each rule is shown in square brackets.

5.4.3 Consulting the Classifier

The same two test instances used for $\mathcal{C}4.5$ will be used here *i.e.*, (outlook, temperature, humidity, windy) = (sunny, 35, 70, no) and (?, 30, 80, yes) — see Section 4.10.2, page 12. In this case, one can use:

```
consultr -f voyage
```

The output of the consultr session is shown in Figure 8.

5.5 URL

You can find $\mathcal{C}4.5$rules in the URL:

http://www.cse.unsw.edu.au/~quinlan/

5.6 Summary

> The most beautiful thing we can experience is the mysterious. It is the source of all true art and Science.
>
> —*Albert Einstein*

The basic steps taken by $\mathcal{C}4.5$rules are:

1. $\mathcal{C}4.5$rules starts extracting rules from the unpruned tree generated by $\mathcal{C}4.5$;
2. each rule is simplified by removing each condition that does not seem to be good for discriminating one class from another using a pessimistic error rate;
3. for each class, an evaluation for all simplified rules for that class is performed to remove rules that do not contribute to accuracy in the remaining rule set;

```
C4.5 [release 8] rule generator Sun Oct 10 01:16:35 1999
-------------------------------

    Options:
        File stem <voyage>

Read 15 cases (4 attributes) from voyage

------------------
Processing tree 0

Final rules from tree 0:

Rule 2:
        outlook = sunny
        humidity > 78
        ->  class dont_go  [63.0%]

Rule 3:
        outlook = rain
        windy = yes
        ->  class dont_go  [50.0%]

Rule 4:
        outlook = rain
        windy = no
        ->  class go  [63.0%]

Rule 1:
        humidity <= 78
        ->  class go  [61.2%]

Default class: go


Evaluation on training data (15 items):

Rule  Size  Error  Used  Wrong            Advantage
----  ----  -----  ----  -----            ---------
   2     2  37.0%     3      0 (0.0%)        3 (3|0)    dont_go
   3     2  50.0%     2      0 (0.0%)        2 (2|0)    dont_go
   4     2  37.0%     3      0 (0.0%)        0 (0|0)    go
   1     1  38.8%     5      0 (0.0%)        0 (0|0)    go

Tested 15, errors 1 (6.7%)   <<


          (a)  (b)      <-classified as
         ---- ----
            9           (a): class go
            1    5      (b): class dont_go
```

Figure 7: The voyage $\mathcal{C}4.5$rules classifier

```
C4.5 [release 8] production rule interpreter    Sun Oct 10 01:21:40 1999
--------------------------------------------

Rule 1:
        outlook = sunny
        humidity > 78
        ->  class dont_go  [63.0%]

Rule 2:
        outlook = rain
        windy = yes
        ->  class dont_go  [50.0%]

Rule 3:
        outlook = rain
        windy = no
        ->  class go  [63.0%]

Rule 4:
        humidity <= 78
        ->  class go  [61.2%]

outlook: sunny
humidity: 70

Decision:
        go  CF = 0.61

Retry, new case or quit [r,n,q]: n

-------------------------------------------

outlook: ?
humidity: 80

Decision:
        go (default class)

Retry, new case or quit [r,n,q]: q
```

Figure 8: Consulting the voyage $\mathcal{C}4.5$rules classifier

4. the rule set for classes are ordered to minimize false positives errors;
5. a default class is chosen.

This process leads to a decision rule classifier that is generally as accurate as the original pruned tree induced by $\mathcal{C}4.5$ but more easier to understand by humans.

6 $\mathcal{C}5.0$

> As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.
>
> —*Albert Einstein*

$\mathcal{C}5.0$ is a commercial version of $\mathcal{C}4.5$. In fact, there are two programs: the $\mathcal{C}5.0$ for Unix platforms and the $\mathcal{S}$ee5 for Windows™. In this work we will refer to $\mathcal{C}5.0$ although the same functionality is embedded in $\mathcal{S}$ee5 using Windows™ graphical components. $\mathcal{C}5.0$ works as its predecessors $\mathcal{C}4.5$ and $\mathcal{C}4.5$rules, but now they are together into a single program. Therefore, we will only report here new features incorporated into $\mathcal{C}5.0$ since its predecessors $\mathcal{C}4.5$ and $\mathcal{C}4.5$rules.

6.1 Program Names

c5.0 This command invokes decision tree building or rule induction, depending on the command line option used (see Section 6.3).

predict After inducing trees or rules with c5.0, this program can be used to consult the generated classifiers. It replaces the consult and consultr tools used previously by $\mathcal{C}4.5$ and $\mathcal{C}4.5$rules, respectively. Valid command line options are:

- `-f` *filestem*
  Specify the file to be read.
- `-p`
  Preview trees or rulesets.
- `-r`
  Use rulesets rather than trees.
- `-h`
  Print help message.

6.2 Files

All files read and written by $\mathcal{C}5.0$ assume the form *filestem.extension*, where *extension* characterizes the type of data involved:

`.names` This text file assumes basically the same format as for $\mathcal{C}4.5$. However, it is possible to create derived features from the original ones as well as ordering of nominal features. When declaring a feature, it is possible to use:

- ignore causing the feature to be completely ignored;
- label serves only to identify a particular instance (but it is not considered when learning concepts). It is used to refer to individual instances. If there are two or more label features, only the last one is used. Observe that this *is not* the class label but purely a mean to differentiate one instance from each other;
- continuous indicating integer or floating point values;
- discrete N where $N > 0$ is an integer specifying the feature can take no more than N discrete values;
- date specifying dates in the form `YYYY/MM/DD`, e.g. 1999/10/25. Valid dates range from the year 1601 to the year 4000;
- a list of names separated by commas, indicating that the feature can take discrete values. One difference from $\mathcal{C}4.5$ is that $\mathcal{C}5.0$ values may be prefixed by [ordered] to indicate that they are given in a meaningful ordering, otherwise they will be taken as unordered. For instance, the values low, medium and high are generally ordered (while male and female are not). The former can be declared as
  temperature: [ordered] low, medium, high.
- it is also possible to define an implicitly-defined feature which is specified by a formula. Each implicitly-defined feature is followed by ':=' and a formula defining the feature value. The formula is written using parentheses when needed, and may refer to any feature defined before this one. Constants in the formula can be numbers (written in decimal notation), dates, and discrete feature values (enclosed in string quotes '"'). The operators and functions that can be used in the formula are

  $+, -, *, /, \%$ (modulus), $\hat{}$ (meaning 'raised to the power')
  $>, >=, <, <=, =, <>$ or $!=$ (both meaning 'not equal')
  and, or (logical operators)
  $\sin(X), \cos(X), \tan(X), \log(X), \exp(X),$ $\text{int}(X)$ (meaning the 'integer part of X')

  Date type can also be used in a formula. In fact, this type of data is such that each date is stored as the number of days since a particular starting date, specifically the date 1600/03/01. Thus, it is possible to do meaningful operations with dates.

  If the value of one formula cannot be determined due to one or more features with unknown values appearing in the formula then the value of the implicitly-defined feature is also unknown.

The order of features declaration is not relevant, but it must follow the exact data position in the `.data` file.

`.data` This file assumes the same format as $\mathcal{C}4.5$. Observe that implicitly-defined features must not be present in this file, since they are calculated by a formula. This file is required for running c5.0.

`.test` This optional file assumes the same form of the `.data` file. It is used to evaluate accuracy of the generated classifier.

`.cases` This file is optional and has exactly the same format as the `.data` file. It differs from the `.test` file only in allowing the instance class labels to be unknown.

`.costs` The costs file is also optional and sets out differential misclassification costs. In some applications there is a much higher penalty for certain types of mistakes. $\mathcal{C}$5.0 allows different misclassification costs to be associated with each combination of predicted class and true class. Each entry has the form:

```
predicted-class, true-class:  cost
```

where `cost` is a non-negative real number. The file, which is automatically read if present, may contain any number of entries; if a particular combination is not specified explicitly, its cost is taken to be 0 if the predicted class is correct and 1 otherwise.

`.tree` Output file containing the final pruned tree. This file is used by predict.

`.rules` Output file containing the final rule set generated by c5.0. It is also used by predict.

6.3 $\mathcal{C}$5.0 Options

This sections describes $\mathcal{C}$5.0 command line options. Note that some options have quite different meaning than in $\mathcal{C}$4.5. For running $\mathcal{C}$5.0, the command line assumes the form:

c5.0 [*options*]

Options can appear in any order and they can be:

- `-f` *filestem*
  Specify the input dataset. No extension should be given. If no *filestem* is given, the default value *DF* is assumed (so, *DF.names* and *DF.data* will be looked in the current path).

- `-b`
  Invoke adaptive boosting: the idea is to generate several classifiers rather than just one. When a new case is to be classified, each classifier votes for its predicted class and the votes are counted to determine the final class.

- `-t` v
  Allow the user to set the number of boosting trials as v.

- `-r`
  Use this option to create rules instead of trees.

- `-u`
  Order rules by utility in bands. When using the `-u` option, by default, $\mathcal{C}$5.0 orders rules by class and sub-orders by confidence. The `-u` option can be selected as an alternative ordering by contribution to predictive accuracy. In this case, the rule that most reduces the error rate appears first and the rule that contributes least appears last. Besides that, results are reported in a selected number of bands so that the predictive accuracies of the more important subsets of rules are also estimated.

- `-p`
  Use soft thresholds. This option enables subsidiary cutpoints (range) for test on continuous features. When this option is invoked, each threshold is broken into three ranges: a

lower bound L, an upper bound U, and a central value T. If the attribute value in question is below L or above U, classification is carried out using the single branch corresponding to the <= or > result respectively. If the value lies between L and U, both branches of the tree are investigated and the results are combined probabilistically. The values of L and U are determined based on an analysis of the apparent sensitivity of classification to small changes in the threshold. They are not necessarily symmetric *i.e.*, a fuzzy threshold can be relatively hard in one direction. The default value is to use hard thresholds (one value).

- `-e`
  Focus on errors instead of on misclassification costs, ignoring the `.costs` file.

- `-s`
  Cause grouping on discrete values (in sets) for node tests. The default is no grouping, so each internal tree node branches separately for every possible feature value.

- `-m` *weight*
  Generally used for noisy data to avoid trees with little predictive power. It indicates that each test must have at least *weight* outcomes with a minimum number of instances (the sum of the weights of the instances for at least two of the subsets must attain some minimum). Higher values may be interesting for tasks with noisy data. The default value is 2.

- `-c` *CF*
  The *CF* value affects tree pruning. Small values cause more heavy pruning than large values. The default value is 25%.

- `-S` *v*
  Indicate the training sample percentage to be used.

- `-I` *v*
  Allow changing the integer random seed to *v*.

- `-h`
  Print the help message.

6.4 Example

In what follows the voyage data (Section 4.10, page 10) will be used to show a simple $\mathcal{C}5.0$ execution.

6.4.1 Input Data

$\mathcal{C}5.0$ can handle $\mathcal{C}4.5$ files, so the same input files used for $\mathcal{C}4.5$ are used also for $\mathcal{C}5.0$. Again, Figures 3 and 4, page 11 show the corresponding `voyage.names` and `voyage.data`, respectively.

6.4.2 The Tree Classifier

For running $\mathcal{C}5.0$ with default options, one just need to specify the *filestem* to be used:

```
C5.0 [Release 1.11]     Sun Oct 31 16:04:32 1999
-------------------
    Options:
        Application 'voyage'

Read 15 cases (4 attributes) from voyage.data

Decision tree:

outlook = overcast: go (5/1)
outlook = sunny:
:...humidity <= 78: go (2)
:   humidity > 78: dont_go (3)
outlook = rain:
:...windy = yes: dont_go (2)
    windy = no: go (3)

Evaluation on training data (15 cases):

          Decision Tree
          ----------------
          Size      Errors

             5    1( 6.7%)   <<


           (a)   (b)    <-classified as
          ----  ----
             9          (a): class go
             1     5    (b): class dont_go

Time: 0.0 secs
```

Figure 9: The voyage $\mathcal{C}5.0$ tree classifier

`c5.0 -f voyage`

Figure 9 shows the output generated by $\mathcal{C}5.0$.

6.4.3 The Rule Classifier

For running $\mathcal{C}5.0$ to induce rules, besides the *filestem*, one just need to specify the `-r` option:

`c5.0 -r -f voyage`

Figure 10 shows the output generated by $\mathcal{C}5.0$. Each rule consists of:

- An arbitrary rule number used only to identify the rule.
- Statistics (p, lift a) or (p/e, lift a) that summarize the performance of the rule, where p is the number of training instances covered by the rule. If e appears, it shows how many of

them do not belong to the class predicted by the rule. The lift a is the estimated accuracy of the rule divided by the prior probability of the predicted class.

- One or more conditions that must be satisfied if the rule is to be applicable.
- A class predicted by the rule.
- A value between 0 and 1 that indicates the confidence with which this prediction is made. If boosting is used, this confidence is measured using an artificial weighting of the training cases and therefore it does not reflect the accuracy of the rule.

One interesting point to be noticed about $\mathcal{C}5.0$ is that all rules are evaluated when classifying a new instance, not just the first which fires. Each fired rule votes for its predicted class with a voting weight equal to its confidence value, the votes are totted up, and the class with the highest total vote is chosen as the final prediction. There is also a default class, that is used when none of the rules apply.

6.4.4 Consulting the Classifier

The same two test instances used for $\mathcal{C}4.5$ will be used here *i.e.*, (outlook, temperature, humidity, windy) = (sunny, 35, 70, no) and (?, 30, 80, yes) — see Section 4.10.2, page 12. In this case, one can use:

```
predict -f voyage
```

to get the answer. The output of the predict session is shown in Figure 11.

In order to predict using rules, one can use:

```
predict -r -f voyage
```

to get the answer. The output of the predict session is shown in Figure 12.

6.5 URL

More information about $\mathcal{C}5.0$ can be found in the URL:

http://www.rulequest.com/

6.6 Summary

Intellectuals solve problems; geniuses prevent them.

—*Albert Einstein*

$\mathcal{C}5.0$ is a commercial product which was evolved from $\mathcal{C}4.5$. It is able to handle ordered nominal features, date features as well as implicitly-defined features. $\mathcal{C}5.0$ is also able to induce rule sets from decision trees.

```
C5.0 [Release 1.11]     Sun Oct 31 16:10:32 1999
-------------------
    Options:
        Application 'voyage'
        Rule-based classifiers
Read 15 cases (4 attributes) from voyage.data

Decision tree:

outlook = overcast: go (5/1)
outlook = sunny:
:...humidity <= 78: go (2)
:   humidity > 78: dont_go (3)
outlook = rain:
:...windy = yes: dont_go (2)
    windy = no: go (3)

Extracted rules:

Rule 1: (3, lift 1.3)
        outlook = rain
        windy = no
        ->  class go  [0.800]
Rule 2: (2, lift 1.2)
        outlook = sunny
        humidity <= 78
        ->  class go  [0.750]
Rule 3: (5/1, lift 1.2)
        outlook = overcast
        ->  class go  [0.714]
Rule 4: (3, lift 2.0)
        outlook = sunny
        humidity > 78
        ->  class dont_go  [0.800]
Rule 5: (2, lift 1.9)
        outlook = rain
        windy = yes
        ->  class dont_go  [0.750]
Default class: go

Evaluation on training data (15 cases):

            Decision Tree           Rules
          ----------------    ----------------
          Size      Errors      No      Errors
             5    1( 6.7%)       5    1( 6.7%)   <<

           (a)   (b)    <-classified as
          ----  ----
             9          (a): class go
             1     5    (b): class dont_go

Time: 0.0 secs
```

Figure 10: The voyage $\mathcal{C}5.0$ rule classifier

```
C5.0 Interactive Classifier [Release 1.11]      Sun Oct 31 16:54:43 1999
------------------------------------------
    Options:
        Application ‘voyage’

outlook: sunny
humidity: 35

    ->  go  [0.75]

Retry, new case or quit [r,n,q]: n

outlook: ?
humidity: 30
windy: yes

    ->  go  [0.49]

Retry, new case or quit [r,n,q]: q
```

Figure 11: Consulting the $\mathcal{C}5.0$ voyage decision tree classifier

```
C5.0 Interactive Classifier [Release 1.11]      Sun Oct 31 17:00:18 1999
------------------------------------------
    Options:
        Use rulesets
        Application ‘voyage’

outlook: sunny
humidity: 35

    ->  go  [0.75]

Retry, new case or quit [r,n,q]: n

outlook: ?

    ->  go  [0.50]

Retry, new case or quit [r,n,q]: q
```

Figure 12: Consulting the $\mathcal{C}5.0$ voyage rule classifier

When classifying a new instance using rule sets, all fired rules are combined to predict the class. As already mentioned, rules are generally much simpler to understand than trees since each rule describes a specific context associated with a class. Furthermore, a rule set generated from a tree usually has fewer rules than the tree has leaves, improving the comprehensibility. However, for very large datasets, generating rules with the rule set option can require considerably more computer time.

7 $\mathcal{OC}1$

> It would have been especially interesting to read what Quinlan thinks of the numerous experiments comparing $\mathcal{C}4.5$ and its predecessors to the other classification algorithms that are commonly used in the research community.
>
> —*Steven L. Salzberg, Book Review of (Quinlan, 1988)*

Oblique Classifier 1 — $\mathcal{OC}1$ — is a decision tree induction system designed for applications where the instances have numeric (continuous) feature values (Murthy et al., 1994). $\mathcal{OC}1$ builds decision trees that contain linear combinations of one or more attributes at each internal node; these trees then partition the space of instances with both oblique and axis–parallel hyperplanes. $\mathcal{OC}1$ has been used for classification of data from diverse problem domains, including astronomy (Salzberg et al., 1995) and DNA sequence analysis (Salzberg, 1995).

Induced trees by $\mathcal{OC}1$ assumes the form shown in Equation 1.

$$a_1\ x[1] + a_2\ x[2] + \ldots + a_m\ x[m] + a_{m+1} = 0 \qquad (1)$$

where a_i is a constant and $x[i]$ is a feature X_i.

7.1 Inducing Trees

The algorithm for inducing axis–parallel considers all possible feature splits and chooses the one that optimizes a predefined goodness measure. Oblique decision tree methods cannot consider all tests due to the increased computational complexity when compared to the axisparallel case.

Considering the complexity of selecting an optimal hyperplane for a single node of a DT, in a domain with n training instances described by m real-valued features, there are nm distinct axis–parallel splits while there are at most $2^m\binom{n}{m}$ oblique splits. In fact, the problem of finding the best oblique split is much more than that of searching the best axis–parallel split *i.e.*, the problem is NP-hard.

In order to be at least as powerful as standard DT methods, $\mathcal{OC}1$ first finds the best axisparallel (univariate) split at a node before looking for an oblique split. Consequently, $\mathcal{OC}1$ uses an oblique split only when it improves over the best axisparallel split.

7.2 Choosing the Best Feature to Split

To find the hyperplane to split the training set T at a node of the decision tree, we can write $Tj = (x_{j1}, x_{j2}, \dots, x_{jd}, C_j)$ for the j-th instance from the training set T. As defined in Equation 1, the equation of the current hyperplane H at a node of the DT is written as $\sum_{i=1}^{m}(a_i x_i) + a_{m+1} = 0$. If we substitute an instance T_j into the equation for H, we get $\sum_{i=1}^{m}(a_i x_{ji}) + a_{m+1} = V_j$ where the sign of V_j tells whether the point T_j is above or below the hyperplane H. If $V_j > 0$, then T_j is above H. If H splits the training set T perfectly, then all points belonging to the same category will have the same sign for V_j *i.e.*, $\text{sign}(V_i) = \text{sign}(V_j)$ if and only if $\text{class}(T_i) = \text{class}(T_j)$.

$\mathcal{OC}1$ adjusts the coefficients of H individually, finding a locally optimal value for one coefficient at a time: each coefficient a_i is treated as a variable, and all other coefficients are treated as constants. Then V_j can be viewed as a function of a_i. In particular, the condition that T_j is above H is equivalent to $V_j > 0$ which is equivalent to Equation 2.

$$a_i > \frac{a_i x_{ji} - V_j}{x_{ji}} = U_j \tag{2}$$

assuming that $x_{ji} > 0$, which is done by normalization. Using the definition of U_j, the instance T_j is above H if $a_i > U_j$ and below otherwise. By plugging all the points from T into Equation 2, n constraints are obtained on the value of a_i. The problem then is to find a value for a_i that satisfies as many of these constraints as possible. This problem is easy to solve optimally sorting all the values U_j and considering a_m the midpoint between each pair of different values.

7.3 Handling Unknown Values

Unknown values are denoted as a question mark. $\mathcal{OC}1$ simply fills all missing values with the mean value of the respective feature.

7.4 Handling Noisy Instances

As most of the decision trees induction algorithms, $\mathcal{OC}1$ uses pruning to handle noise. Specifically, $\mathcal{OC}1$ uses the Cost Complexity — CC — pruning method described in (Breiman et al., 1984). This method (also known as *error complexity* or *weakest link* pruning) requires a separate pruning set. $\mathcal{OC}1$ randomly chooses 10% of the training data to use for pruning.

The basic idea behind CC pruning is to create a set of trees of decreasing size from the original, complete tree. All these trees are used to classify the pruning set, and accuracy is estimated from those. CC pruning then chooses the smallest tree whose accuracy is within s standard errors squared of the best accuracy obtained. When the 0-SE rule ($s = 0$) is used, the tree with highest accuracy on the pruning set is selected. When $s > 0$, smaller tree size is preferred over higher accuracy.

7.5 Classifying New Instances

As any DT algorithm $\mathcal{OC}1$ begins from the root of the induced tree, tests-and–branches each node with the respective feature until it reaches one leaf. The class prediction of this instance is then assigned as the class of that leaf.

7.6 Files

All files read and written by $\mathcal{OC}1$ assume the form *filestem.extension*, where *extension* characterizes the type of data involved. Differently of the other inducers, $\mathcal{OC}1$ does not require a `.names` file since all features must be continuous. Features are simply named as $x[1], x[2], \ldots, x[m]$ according with the order they appear in the `.data` file.

`.data` Each line from this text file describes one instance, providing values for each feature. Numeric attributes are listed before the integer class label and each value can be separated by blank spaces, tabs or commas. Unknown values can be specified by a question mark.

`.test` Optional text file which assumes the same format of the `.data` file. It is used to evaluate accuracy of the generated classifier.

`.dt` Output file containing pruned trees generated by $\mathcal{OC}1$.

`.unpruned` Output file containing unpruned trees generated by $\mathcal{OC}1$.

`.classified` Output file which is created by $\mathcal{OC}1$ when classifying unlabeled instances (see `-u option` in the next section).

`.misclassified` Output file which is created by $\mathcal{OC}1$ when classifying labeled instances. This file contains only those instances that are incorrectly classified (see `-M` option in the next section).

7.7 $\mathcal{OC}1$ Options

For running $\mathcal{OC}1$, the command line assumes the form:

mktree [*options*]

Options to $\mathcal{OC}1$ have the following meanings:

- `-a`
  Consider only axis parallel splits at each node of the DT. With this option, $\mathcal{OC}1$ builds trees in the normal way. The main use of this option is a faster construction of the DT.

- `-A`*filestem*
  Indicate the file where to output the animation information. The default is no output. If a file name is specified with this option, and if no cross–validation is used, and a training set is specified, $\mathcal{OC}1$ outputs all hyperplane perturbations constructed into file *filestem*. This file can be used with the `-D` option of the display tool to produce a PostScript pseudo-animation file.

- `-b`v
  Bias towards axis parallel splits if $v \geq 1.0$. The default is $v = 1.0$. At any node of the decision tree, an oblique split is preferred to an axis–parallel split only if the ratio of the axis parallel impurity to the oblique impurity is greater than v.

- `-B`
  Change the order of coefficients perturbation to Best–First. At each node of the tree, $\mathcal{OC}1$ tries to perturb (adjust) the coefficients of the hyperplane it is considering at one time. This option tells $\mathcal{OC}1$ to perturb first the coefficient that gives the greatest improvement in the impurity measure. The default is the sequential order which means that $\mathcal{OC}1$ will cycle through all the coefficients of a hyperplane in order, perturbing each one to the best value it can find for that coefficient.

- `-c`k
  Allow defining the number of classes as k. The default is to compute from data or decision tree. If a decision tree is provided with the `-D` option, or if a training set is specified with the `-t` option, $\mathcal{OC}1$ computes the number of categories and the number of attributes automatically. Class labels have to be integers.

- `-d`m
  Define the number of features as m. The default is to compute m from the data or the decision tree. If a decision tree is provided with the `-D` option, or if a training set is specified with the `-t` option, $\mathcal{OC}1$ computes the number of categories and the number of attributes automatically. The default is the number of white-space or comma separated real numbers in line 1 of the training data file minus 1. The last field is assumed to be the category of the instance.

- `-D`*filestem*
  Define the decision tree output file. The default is *filestem*`.dt`. If a training set is specified with the `-t` option, then $\mathcal{OC}1$ builds a tree and writes it to this file. If no training set is specified, then $\mathcal{OC}1$ assumes using an existing tree to classify or estimate the accuracy on some data, and the decision tree is read from this file. The output file names are *filestem*`.dt` for the pruned tree and *filestem*`.unpruned` for the unpruned tree. If no pruning is chosen (with the `-p0` option), or if no pruning is possible on the induced tree, the unpruned tree is written to *filestem*. If cross validation is chosen (`-V` option), the decision tree corresponding to the first fold is written to file.

- `-i`v or `-r`v
  Define the number of restarts/iterations for the perturbation algorithm. The default is $v = 20$ which means that, at every node of the decision tree, $\mathcal{OC}1$ starts with the best axis parallel split (if `-o` option is not chosen) and $(v - 1) = 19$ different random hyperplanes, perturb each one of them to the best position possible, and chooses the best of the resulting $v = 20$ hyperplanes. Choosing a larger value of v will slow down the program, but can result in better trees, since the program searches more at each node for a good hyperplane.

- `-j`v or `-m`v
  Define the maximum number of random jumps tried at each local minimum. The default value is $v = 5$. When $\mathcal{OC}1$ cannot improve any of the coefficients of a hyperplane deterministically, it is stuck in a local minimum. It then attempts to jump out of this minimum by choosing a random direction, and sliding the hyperplane in that direction. Setting $v = 5$ means that $\mathcal{OC}1$ tries at most 5 times to jump in a random direction and it returns to the deterministic perturbation algorithm as soon as it finds an improvement in the impurity measure.

- `-K`
  Use CART–linear combinations mode. If this mode is chosen, $\mathcal{OC}1$ suspends its own coefficient perturbation algorithm, and executes CART's deterministic perturbation algorithm.

Note that the CART mode does not include backward feature elimination and normalization (zero mean, unit standard deviation) at each tree node. This mode is intended only to contrast the coefficient perturbation methods of $\mathcal{OC}1$ and CART-LC *i.e.*, CART with linear combinations (Breiman et al., 1984, Chapter 5).

- `-l`*filestem*
  Define the log file. The default is `oc1.log`.

- `-M`*filestem*
  Define the file to output misclassified instances. The default is no output. Users might want to know which instances are misclassified in an experiment, to gain insights into, say, the particular attributes being used. If this option is specified, $\mathcal{OC}1$ writes all misclassified instances to the specified file.

- `-n`v
  Define the number v of training instances. The default is all n instances in the training file (given with the `-t` option). If this number is less than the total instances n in the file specified with the `-t` option, then $\mathcal{OC}1$ randomly chooses v data points for the training set and the rest of the file for the test set. If, however, `-T` option is chosen, all instances in the training file are used for learning.

- `-N`
  Tell $\mathcal{OC}1$ not to apply normalization at each tree node. $\mathcal{OC}1$'s hill-climbing coefficient perturbation algorithm requires that all attribute values be positive. So, at each tree node, data is translated into the positive quadrant as a normalization step, and the hyperplane induced is unnormalized subsequently. This option enables the user to switch off this normalization.

- `-o`
  Force $\mathcal{OC}1$ to use only oblique splits.

- `-p`v
  Define the proportion of training set to be used in pruning. The default is $v = 0.1$. $\mathcal{OC}1$ always prunes decision trees by default, to avoid the problem of overfitting. Currently, the only pruning method implemented is error complexity pruning using a separate pruning set. This option specifies to use a randomly chosen set of vn (where $n =$ training set size) points from the training set exclusively for pruning. If $v = 0$, no pruning is done.

- `-R`v
  Set the order of coefficients perturbation to random and perturb v times. Thus, if `-R50` is chosen, $\mathcal{OC}1$ will repeat the following loop 50 times: pick a random coefficient and attempt to perturb it. There is no default value.

- `-s`v
  Set the integer seed for the random number generator to v. The default is the operating system srand48() system call value.

- `-t`*filestem*
  Define the file containing training instances. There is no default value. If no file name for test data is specified (with the `-T` option), and if the number of training instances given with the `-n` option is less than the number of instances in the above file, then test data is also read from this file.

- `-T`*filestem*
  Define the file containing test instances. There is no default value. This option overrides the `-n` option and it is overridden by the `-V` option.

- `-u`
  Specify the test data is unlabeled and that should be labeled by $\mathcal{OC}1$. The data is classified using the decision tree which is in turn read from a file or induced from the training data. The classified data is written to `.classified` file.

- `-v`
  Set the verbosity level if specified once and increases this level if specified more than once. The default is no verbosity.

- `-V`v
  Define the number of partitions for cross–validation as v. The default is $v = 0$ (no cross-validation). For $v = -1$ leave–one–out is performed.

  If a number v is specified, the data are divided randomly into v partitions. If v does not evenly divide the data into the number of instances, then the extra instances are put in the last partition. Each partition is used once as a test set for a tree built from the remaining instances. Leave-one-out is equivalent to setting $v =$ total number of instances.

7.8 Example

In what follows the voyage data (Section 4.10, page 10) will be used to show a simple $\mathcal{C}5.0$ execution.

7.8.1 Input Data

Figure 13 shows the corresponding `voyage.data`. Observe that, since $\mathcal{OC}1$ is not able to deal with nominal features they have been transformed into binary values — see (Weiss and Indurkhya, 1998) for more details about converting features. The feature outlook was transformed into 3 columns assuming boolean values, the first column is 1 for "sunny" and 0 otherwise. The second assumes 1 for "overcast" and 0 otherwise and, finally, the third column assumes 1 for "rain" and 0 otherwise.

The feature wind (column six) was mapped 0 for "no" and 1 for "yes". The class label was also mapped 0 for "dont_go" and 1 for "go" class labels.

7.8.2 The Classifier

For running $\mathcal{OC}1$ with default options, we just need to specify the *filestem* to be used:

```
mktree -tvoyage.data
```

Figure 14 shows the output generated by $\mathcal{OC}1$. Observe that the class label "0" was remapped to "2" by $\mathcal{OC}1$. Figure 15 shows the unpruned tree induced by $\mathcal{OC}1$.

```
1 0 0    25 72   1      1
1 0 0    28 91   1      0
1 0 0    22 70   0      1
1 0 0    23 95   0      0
1 0 0    30 85   0      0
0 1 0    23 90   1      1
0 1 0    29 78   0      1
0 1 0    19 65   1      0
0 1 0    26 75   0      1
0 1 0    20 87   1      1
0 0 1    22 95   0      1
0 0 1    19 70   1      0
0 0 1    23 80   1      0
0 0 1    25 81   0      1
0 0 1    21 80   0      1
```

Figure 13: The $\mathcal{OC}1$ `voyage.data` file

```
Remapping class numbers:
          0 To   2
Pruned decision tree written to:   voyage.dt
Unpruned decision tree written to: voyage.unp

acc. on training set = 86.67    #leaves = 2     max depth = 1

Training set: voyage.data, Dimensions: 6, Categories: 2

Root Hyperplane: Left = [7,0], Right = [2,5]
16.814833 x[1] + -1.860402 x[2] + -0.995862 x[3] + 0.253328 x[4] +
-0.089471 x[5] +  2.365013 x[6] +  0.995192 = 0
```

Figure 14: The voyage $\mathcal{OC}1$ classifier

```
Training set: voyage.data, Dimensions: 6, Categories: 2

Root Hyperplane: Left = [7,0], Right = [2,5]
16.814833 x[1] + -1.860402 x[2] + -0.995862 x[3] + 0.253328 x[4] +
-0.089471 x[5] + 2.365013 x[6]  + 0.995192 = 0

r Hyperplane: Left = [0,3], Right = [2,2]
1.000000 x[1] + -0.500000 = 0

rr Hyperplane: Left = [2,0], Right = [0,2]
1.000000 x[5] + -81.500000 = 0
```

Figure 15: The voyage $\mathcal{OC}1$ unpruned classifier

```
1.000000  0.000000 0.000000 35.000000 70.000000 0.000000       2
Test instances with labels written to voyage.unl.classified.
```

Figure 16: Consulting the $\mathcal{OC}1$ voyage classifier

7.8.3 Consulting the Classifier

Just one of the two test instances used before will be used here *i.e.*, (outlook, temperature, humidity, windy) = (sunny, 35, 70, no) — see Section 4.10.2, page 12. The other instance (?, 30, 80, yes) was not used since it contains multiple unknown values that are not treated by $\mathcal{OC}1$ [1]. In this case, the unlabeled instance was written to file `voyage.unl` without the label in the form:

```
1 0 0    35 70   0
```

For labeling data, the following command was used:

```
mktree -Dvoyage.dt -uTvoyage.unl
```

Note that is necessary to supply the decision tree file to be used, in this case `voyage.dt`. The output of this command is shown in Figure 16. From this output, the predicted label is "2" which means "dont_go" in the original dataset (remember automatic remapping done by $\mathcal{OC}1$ from label "0" to "2").

7.9 URL

You can find more information about $\mathcal{OC}1$ in the URL:

http://www.tigr.org/~salzberg/announce-oc1.html

7.10 Summary

When you stop learning, stop listening, stop looking and asking questions, always new questions, then it is time to die.

—*Lillian Smith*

$\mathcal{OC}1$ is written in ANSI C, and incorporates a number of features intended to support flexible experimentation on a variety of types of data, including cross–validation, generation of artificial data as well as graphical display of data sets and decision trees. The $\mathcal{OC}1$ software allows the user to create both standard, axis–parallel decision trees and oblique (multithetic) trees. However, all features must be continuous since $\mathcal{OC}1$ cannot handle nominal ones restricting its use in some domains in which it will be required some pre-processing in the data to meet this restriction.

2

[1] In the tests performed, $\mathcal{OC}1$ accepts just one unknown value at each unlabeled (new) instance.

8 $\mathcal{CN}2$

In the middle of difficulty lies opportunity.

—Albert Einstein

$\mathcal{CN}2$ is a machine learning algorithm that induces '*if* <complex> *then* <class>' rules in domains where there might be noise. The $\mathcal{CN}2$ algorithm consists of two basic procedures: a search mechanism that performs a beam search for a good rule and a control mechanism for repeatedly executing this search. $\mathcal{CN}2$ is able to generate ordered or unordered rules. The next sections show some basics characteristics of each one.

8.1 Ordered $\mathcal{CN}2$

The original $\mathcal{CN}2$ algorithm induces ordered rules and is fully described in (Clark and Niblett, 1987). The remaining subsections describe the ordered $\mathcal{CN}2$— see Algorithm 1. Observe that Algorithm 3 (page 40) is used for both ordered and unordered rule induction.

8.1.1 Inducing Rules

In the original version of $\mathcal{CN}2$, an ordered list of classification rules is induced from instances using entropy as search heuristic.

In the learning step, $\mathcal{CN}2$ works in an iterative fashion, each iteration searches for a <complex> covering a large number of instances of a single <class = C_i> and few of other classes. The <complex> is both predictive and reliable, as determined by $\mathcal{CN}2$'s evaluation functions. Having found a good <complex>, *all those instances it covers* are removed from the training set and the rule

$$\textit{if} <\text{complex}> \textit{then} <\text{class} = C_i> [\#C_1, \#C_2, \ldots, 0, \#C_i, 0, \ldots, \#C_k]$$

is added to the end of the rule list. This process iterates until no more satisfactory <complex> can be found. Numbers between brackets $\#C_j$ indicate the number of instances covered by the rule for each class C_j. Class order is the same as declared in the `.att` file. In fact, if the induced rule just covers 100% of instances from class C_i which satisfies its <complex>, it would look like:

$$\textit{if} <\text{complex}> \textit{then} <\text{class} = C_i> [0, 0, \ldots, 0, \#C_i, 0, \ldots, 0]$$

8.1.2 Handling Continuous Values

$\mathcal{CN}2$ deals with continuous features by dividing the range of values of each feature into discrete sub-ranges. Tests on such features assume the form:

$$X_i \ \textit{op} \ \textit{Value}$$

where X_i is a feature, *op* is an operator in the set $\{<, \leq, >, \geq\}$ and *Value* is a valid value for feature X_i determined by $\mathcal{CN}2$.

Algorithm 1 $\mathcal{CN}2$ Ordered rule induction

```
1: procedure CN2_ordered_rules(Instances)
2: Rulelist := {}
3: repeat
4:    best_conjunct := find_best_conjunction(Instances)
5:    if best_conjunct ≠ {} then
6:       C := most frequent class in Instances satisfying best_conjunct
7:       Rule := IF best_conjunct THEN class = C
8:       Rulelist := Rulelist + {Rule}
9:       Instances := Instances − {T : T in Instances and satisfies(T,best_conjunct)}
10:   end if
11: until best_conjunct = {}
12: return Rulelist
```

8.1.3 Handling Unknown Values

For unknown feature values, $\mathcal{CN}2$ uses the method of simply replacing unknown values with the most commonly occurring value if the feature is nominal. For continuous features, the midvalue of the most commonly occurring sub-range replaces the unknown value.

8.1.4 Handling Noisy Instances

$\mathcal{CN}2$ embeds an evaluation function that tests whether a <complex> is significant or not due to noisy data. A significant <complex> locates a regularity unlikely to have occurred by chance, thus reflecting a good correlation between feature values and classes. To assess significance, $\mathcal{CN}2$ compares the observed distribution among classes of instances satisfying the <complex> with the expected distribution that would result if the <complex> selects instances randomly. Some differences in these distributions will result from random variation. If the observed differences are too great to be accounted purely by chance, $\mathcal{CN}2$ assumes that the <complex> reflects a good correlation between features and classes.

8.1.5 Classifying New Instances

To classify new instances, using the induced rules, the original $\mathcal{CN}2$ applies an interpretation in which each rule is tried in order until it is found one whose conditions are satisfied by the new instance being classified. The resulting class prediction of this rule is then assigned as the class of that instance. Thus the *order* of rules is important. If no rule is satisfied, the default rule assigns the most common class in the training set to the new instance.

With ordered $\mathcal{CN}2$ rules a similar policy to unordered rules (see below) is followed for handling unknowns, except after a rule has fired absorbing, say, 1/3 of the testing instance, only the remaining 2/3 are sent down the remainder of the rule list. The first rule will cause $1/3\times$ class distribution to be collected, but a second similar rule will cause $2/3 \times 1/3\times$ class distribution to be collected. Thus the 'fraction' of the instance gets less and less as it progresses down the rule list.

8.2 Unordered $\mathcal{CN}2$

An improved version of $\mathcal{CN}2$, which induces unordered rules, is described in (Clark and Niblett, 1989; Clark and Boswell, 1991). The remaining subsections describe the unordered $\mathcal{CN}2$

Algorithm 2 presents the $\mathcal{CN}2$ unordered algorithm and Algorithm 3, as stated earlier, allows $\mathcal{CN}2$ finding the best conjunction at each step for both ordered and unordered rule induction.

8.2.1 Inducing Rules

Some improvements in $\mathcal{CN}2$ are described in (Clark and Boswell, 1991) where the Laplacian error estimate is used as an alternative evaluation function and — besides ordered rules — the algorithm can generate unordered rules as well.

The main modification to the original algorithm — in order to generate unordered rules — is to iterate the search class in turn, removing *only the covered instances of that class* when a rule has been found. Unlike the ordered rules, the negative instances remain because now each rule must independently stand against all negatives. The covered positives instances must be removed to prevent $\mathcal{CN}2$ repeatedly finding the same rule.

8.2.2 Classifying New Instances

To classify a new instance using induced unordered rules, all rules are tried and those which fire are collected. If more than one class is predicted by fired rules, the method used is to tag each rule with the distribution of covered instances among classes and then to sum these distributions to find the most probable class. For instance, consider the three rules:

if	head=square	**and**	hold=gun	**then**	class=enemy	covers	[15,1]
if	size=tall	**and**	flies=no	**then**	class=friend	covers	[1,10]
if	look=angry			**then**	class=enemy	covers	[20,0]

Here the two classes are [enemy,friend] and [15,1] denotes that the rule covers 15 training instances of enemy and 1 of friend. Given a new instance of a robot which has square head, carries a gun, tall, non-flying and angry, all three rules are fired. $\mathcal{CN}2$ resolve this clash by summing the covered instances [36,11] and then predicting the most common class in the sum — enemy.

8.2.3 Handling Unknown Values

During rule induction, unknown values are split into fractional instances. In this case, the counts attached to rules when writing the ruleset should be those encountered during rule generation. However, for unordered rules, the counts to attach are generated after rule generation in a second pass, following the execution policy of splitting an instance with unknown feature value into equal fractions for each value, rather than the Laplace-estimated fractions used during rule generation.

Algorithm 2 $\mathcal{CN}2$ Unordered rule induction

```
1:  procedure CN2_unordered_rules(Instances)
2:  Ruleset := {}
3:  for all class C in Instances do
4:     Rules_for_one_class := {}
5:     repeat
6:        best_conjunct := find_best_conjunction(Instances,C)
7:        if best_conjunct ≠ {} then
8:           Rules_for_one_class := Rules_for_one_class + {IF best_conjunct THEN class = C}
9:           Instances := Instances − {T : T in Instances and satisfies(T,best_conjunct and
             class=C)}
10:       end if
11:    until best_conjunct = {}
12:    Ruleset := Ruleset + Rules_for_one_class
13: end for
14: return Ruleset
```

8.2.4 Handling Noisy Instances

$\mathcal{CN}2$ unordered rules treats noisy data just like $\mathcal{CN}2$ ordered rules does.

8.3 Program Name

cn This command invokes rule induction (see Section 8.5 for options of this command).

8.4 Files

All files read and written by $\mathcal{CN}2$ assume the form *filestem.extension*. However, unlike other systems, *extension* may be any string the user chooses. In general, people working with $\mathcal{CN}2$ follow the nomenclature for naming files as suggested by $\mathcal{CN}2$ authors:

`.att` This text file provides names for classes, features and enumerates possible feature values. Each line ending with a semicolon. Blank lines, spaces and tabs can be used for better clarity. Anything from the percent symbol `%` to the end of line is ignored and can be used for comments.

The first line must be the token `**ATTRIBUTE FILE**`; remaining entries consist of one line for each feature. A feature description begins with the feature name (no comma, colon, vertical bar and backslash are allowed in feature names) followed by a colon and then possible values that feature can take:

- (INT) indicating integer values;
- (FLOAT) indicating floating point values;
- a list of names separated by spaces or tabs, indicating that the feature can take discrete values.

The order of feature declaration is not relevant, but it must follow the exact data position in the `.exs` file.

Algorithm 3 $\mathcal{CN}2$ Find best conjunction

```
1: procedure CN2_find_best_conjunction(Instances,[class])
2: most_general_condition := true
3: star := {most_general_condition}
4: best_condition := {}
5: while star ≠ {} do
6:   newstar := {}
7:   for all condition in star do
8:     for all possible feature test not already tested on condition do
9:       cond := condition + {and test}
10:      if cond is better than best_condition and cond is statistically significant then
11:        best_condition := cond
12:      end if
13:      newstar := newstar + cond
14:      if size(newstar) > maxstar then
15:        newstar := newstar − {the worst condition in newstar}
16:      end if
17:    end for
18:  end for
19:  star := newstar
20: end while
21: return best_condition
```

The last column indicates the class. Class labels must end with a semicolon. There must be at least two class names, no matter their order.

It is possible to impose constraints on the order in which features are tested in rule conditions. Such ordering may be optionally imposed on $\mathcal{CN}2$ after feature declarations. A feature ordering takes the form:

$$X_i \text{ BEFORE } X_j;$$

where X_i and X_j are features. Many such declarations can be used. The only effect of this declaration, for instance, sex BEFORE pregnancy is to ensure that any rule containing a test for pregnancy also contains a test of sex.

`.exs` The first entry must be the token `**EXAMPLE FILE**`. Each line describes one instance, providing values for each feature. Each value must be separated by spaces or tabs and terminated by a semicolon. Feature values must appear exactly in the same order as they are declared in the `.att` file. Unknown values can be specified by a question mark (?).

It is possible to assign weights for instances. In this case, besides feature values, each entry can be followed by a `w` W where $W > 0$. When not user supplied, $\mathcal{CN}2$ assumes $W = 1$; otherwise W indicates how many times that instance should be considered. For example, an instance in this file usually is represented as:

sunny 25 72 yes go;

which is equivalent to:

sunny 25 72 yes go **w 1**;

If one wishes to considers this same instance 3 times, it would look like:

sunny 25 72 yes go **w** **3**;

`.tst` The first line of this file must be the token `**EXAMPLE FILE**`. This optional file assumes the same form of the `.exs` file. It is used to evaluate accuracy of the generated classifier.

`.rules` The first line is the token `**RULE FILE**`. This is an output file containing the final rule set generated by $\mathcal{CN}2$.

8.5 $\mathcal{CN}2$ Options

This section describes the $\mathcal{CN}2$ top level menu options. In $\mathcal{CN}2$ each command is invoked by typing its initial letter (upper or lower case), except for `eXecute`, which is invoked by `x`. The behaviour of $\mathcal{CN}2$ can be modified by changing various options. However, all these parameters have default values set at startup, so there is no need to set them before processing data.

- `Read`
  Read a feature (attribute), example (instance), attribute-and-example or rule file. It is important to know that when loading several files in succession, $\mathcal{CN}2$ can only retain in memory one set of attributes, and one set of examples, at any one time. Therefore:

  1. it is necessary to load the appropriate feature file before loading an instance or rule file;
  2. loading a file of any type overwrites any data of that type which may have been loaded previously. Indeed, loading a set of features causes any previously loaded instances to be lost, even if the load fails.

- `Induce`
  Run $\mathcal{CN}2$ on the most recently read set of features and instances.

- `Write`
  After using the `Induce` option, it is possible to use this option to write the rule set to a file.

- `Ckrl`
  Write data in CKRL format, used in the Turing Institute, at Glasgow, Escocia where $\mathcal{CN}2$ was implemented.

- `eXecute`
  Enable the *evaluation* mode. The performance of the current rules may be assessed with respect to the current examples. In this mode, valid options are:

  1. `All`: Evaluate the rule set as a whole.
  2. `Individual`: Evaluate rules individually.
  3. `Help/?`: Display a menu of commands.
  4. `<RET>`: Return to the $\mathcal{CN}2$ prompt.

- `Algorithm`
  Specify whether $\mathcal{CN}2$ is to produce ordered or unordered sets of rules.

- `Error estimate`
  Define whether $\mathcal{CN}2$ should use the Laplacian or the naïve estimate to assess the accuracy of a rule.
- `Star size`
  Query or change the star size.
- `Threshold`
  Query or change the significance threshold.
- `Display`
  Set the verbosity level of $\mathcal{CN}2$ about the search. By default, $\mathcal{CN}2$ just displays each rule as it finds it, but if required $\mathcal{CN}2$ can indicate whenever a new best node is found, or which node is currently being specialized. This option can be used to clarify what $\mathcal{CN}2$ is doing, particularly if it is generating unexpected answers.
- `Help/?`
  Display a menu of commands.
- `Quit`
  Quit $\mathcal{CN}2$.

8.6 Batch Mode

It is possible to run $\mathcal{CN}2$ noninteractively by supplying it with a sequence of options in a file. In this case, $\mathcal{CN}2$ will write a trace of its activities to the standard output. Consequently, if the user wants to run it in background, s/he should redirect its output to a file.

Commands should be present in the command file like as they would be typed in the interactive prompt. In the case of single-character commands, the complete command may be used to make the file more readable. Each command should be separated from the next item with some spaces. Filenames must be followed by a newline. Characters from '#' to the next end-of-line will be regarded as comments and ignored, as they normally are in C shell scripts.

8.7 Example

In what follows the voyage data (Section 4.10, page 10) will be used to show a simple $\mathcal{CN}2$ execution.

8.7.1 Input Data

Figures 17 and 18 show the corresponding `voyage.att` and `voyage.exs`, respectively. For this example, no `voyage.test` file is used.

8.7.2 The Ordered Rules Classifier

Similarly to unordered rules, we run $\mathcal{CN}2$ ordered rules in batch mode through the batch file described in Figure 19. $\mathcal{CN}2$ is then activated by the following command:

```
**ATTRIBUTE FILE**

% Features
outlook:     sunny overcast rain;
temperature: (INT)
humidity:    (FLOAT)
windy:       yes no;

% Class
class:       go dont_go;
```

Figure 17: The $\mathcal{CN}2$ voyage.att file

```
**EXAMPLE FILE**

% Instances
sunny           25      72      yes     go;
sunny           28      91      yes     dont_go;
sunny           22      70      no      go;
sunny           23      95      no      dont_go;
sunny           30      85      no      dont_go;
overcast        23      90      yes     go;
overcast        29      78      no      go;
overcast        19      65      yes     go;
overcast        26      75      no      go;
overcast        20      87      yes     dont_go;
rain            22      95      no      go;
rain            19      70      yes     dont_go;
rain            23      80      yes     dont_go;
rain            25      81      no      go;
rain            21      80      no      go;
```

Figure 18: The $\mathcal{CN}2$ voyage.exs file

```
cn < voyage.cn2.ordered-commands > voyage.cn2.ordered-run.out
```

Figure 20 shows the output generated by ordered $\mathcal{CN}2$.

8.7.3 The Unordered Rules Classifier

One can run $\mathcal{CN}2$ with default options just by typing commands to read the feature and instance files before inducing rules or by using the batch mode. In what follows we use the batch mode to run $\mathcal{CN}2$ through the batch file described in Figure 21. $\mathcal{CN}2$ is then activated by the following command:

```
cn < voyage.cn2.unordered-commands > voyage.cn2.unordered-run.out
```

Figure 22 shows the output generated by $\mathcal{CN}2$.

```
Read atts voyage.att
Read exs voyage.exs
Algorithm
Ordered
Induce
Xecute
All
Quit
Write
voyage.cn2.ordered-rules.out
Quit
```

Figure 19: The `voyage.cn2.ordered-commands` batch file

8.7.4 Consulting the Classifier

The same two test instances used for $\mathcal{C}4.5$ will be used here *i.e.*, (outlook, temperature, humidity, windy) = (sunny, 35, 70, no) and (?, 30, 80, yes) — see Section 4.10.2, page 12.

Since $\mathcal{CN}2$ does not provide a tool for predicting the class for unlabeled instances, one can override this situation putting each unlabeled instance into separated files and 'guessing' a label for each one. After that, we can compare the predicted versus actual labels.

Considering the example above, one instance was written in the `voyage.tst1` file in the form:

```
**EXAMPLE FILE**

% Instance 1
sunny           35      70      no      dont_go;
```

and the other instance in the `voyage.tst2` file:

```
**EXAMPLE FILE**

% Instance 2
?               30      90      yes     dont_go;
```

both with `dont_go` label. After that, we ran $\mathcal{CN}2$ using ordered and unordered rules, as shown in Figures 23 and 24, respectively. Observe in Figure 23 that both instances were classified as `go`, as shown by each confusion matrix. In Figure 24 the first instance was classified as `go` and the second one was classified as `dont_go`.

8.8 URL

One can find more information about $\mathcal{CN}2$ in the URL:

http://www.cs.utexas.edu/users/pclark/software.html

```
            ********************************
            *                              *
            *        Welcome to CN2!       *
            *                              *
            ********************************
CN> Read
READ> Both, Atts, Examples or Rules? Attributes
READ> Filename? voyage.att
Reading attributes...
Finished reading attribute file!
CN> Read
READ> Both, Atts, Examples or Rules? Examples
READ> Filename? voyage.exs
Reading examples...
15 examples!
Finished reading example file!
CN> Algorithm
Algorithm is currently set to UN_ORDERED
ALGORITHM> Ordered
  CN will produce an ordered rule set
CN> Induce
Running CN on current example set...
     *-----------------------*
     |   ORDERED RULE LIST   |
     *-----------------------*
IF    humidity < 83.00
  AND windy = no
THEN  class = go  [5 0]
ELSE
IF    76.00 < humidity < 88.50
THEN  class = dont_go  [0 3]
ELSE
IF    outlook = overcast
THEN  class = go  [2 0]
ELSE
IF    outlook = sunny
  AND humidity > 81.50
THEN  class = dont_go  [0 2]
ELSE
IF    temperature > 20.50
THEN  class = go  [2 0]
ELSE
(DEFAULT) class = dont_go  [0 1]
CN> Execute
EVAL> all
Executing rules...
        PREDICTED
ACTUAL     go      dont_go Accuracy
      go     9       0     100.0 %
 dont_go     0       6     100.0 %
Overall accuracy: 100.0 %
EVAL>
CN> Write
WRITE> Filename? voyage.cn2.ordered-rules.out
Writing current rules to
        voyage.cn2.ordered-rules.out
CN> Quit
Have a nice day!
```

Figure 20: The voyage $\mathcal{CN}2$ ordered rule classifier output

```
Read atts voyage.att
Read exs voyage.exs
Induce
Xecute
All
Quit
Write
voyage.cn2.unordered-rules.out
Quit
```

Figure 21: The voyage.cn2.unordered-commands batch file

```
           *******************************
           *                             *
           *        Welcome to CN2!      *
           *                             *
           *******************************

CN> Read
READ> Both, Atts, Examples or Rules? Attributes
READ> Filename? voyage.att
Reading attributes...
Finished reading attribute file!
CN> Read
READ> Both, Atts, Examples or Rules? Examples
READ> Filename? voyage.exs
Reading examples...
15 examples!
Finished reading example file!
CN> Induce
Running CN on current example set...

     *------------------------*
     |  UN-ORDERED RULE LIST  |
     *------------------------*

IF    humidity < 83.00
  AND windy = no
THEN  class = go  [5 0]

IF    humidity < 67.50
THEN  class = go  [1 0]

IF    24.00 < temperature < 26.50
THEN  class = go  [3 0]

IF    outlook = rain
  AND humidity > 87.50
THEN  class = go  [1 0]

IF    outlook = overcast
  AND humidity > 88.50
THEN  class = go  [1 0]

IF    outlook = sunny
  AND humidity > 78.50
THEN  class = dont_go  [0 3]

IF    temperature < 24.00
  AND 67.50 < humidity < 88.50
  AND windy = yes
THEN  class = dont_go  [0 3]

(DEFAULT) class = go  [9 6]

CN> Execute
EVAL> all
Executing rules...
        PREDICTED
ACTUAL     go      dont_go Accuracy
      go     9       0     100.0 %
 dont_go     0       6     100.0 %
Overall accuracy: 100.0 %
Default accuracy:  60.0 %
EVAL>
CN> Write
WRITE> Filename? voyage.cn2.unordered-rules.out
Writing current rules to
        voyage.cn2.unordered-rules.out
CN> Quit
Have a nice day!
```

Figure 22: The voyage $\mathcal{CN}2$ unordered classifier

```
            *******************************
            *                             *
            *        Welcome to CN2!      *
            *                             *
            *******************************

CN> Read
READ> Both, Atts, Examples or Rules? Attributes
READ> Filename? voyage.att
Reading attributes...
Finished reading attribute file!
CN> Read
READ> Both, Atts, Examples or Rules? Rules
READ> Filename? voyage.cn2.ordered-rules.out
Reading rules...
Finished reading rules!
CN> Read
READ> Both, Atts, Examples or Rules? Examples
READ> Filename? voyage.tst1
Reading examples...
1 examples!
Finished reading example file!
CN> Execute
EVAL> all
Executing rules...
        PREDICTED
ACTUAL     go      dont_go Accuracy
      go     0       0    ---
 dont_go     1       0       0.0 %
Overall accuracy:   0.0 %

EVAL>
CN> Read
READ> Both, Atts, Examples or Rules? Examples
READ> Filename? voyage.tst2
Reading examples...
1 examples!
Finished reading example file!
CN> Execute
EVAL> all
Executing rules...
        PREDICTED
ACTUAL     go      dont_go Accuracy
      go     0       0    ---
 dont_go     1       0       0.0 %
Overall accuracy:   0.0 %
EVAL>
```

Figure 23: Consulting the voyage $\mathcal{CN}2$ ordered classifier

```
            ********************************
            *                              *
            *        Welcome to CN2!       *
            *                              *
            ********************************

CN> Read
READ> Both, Atts, Examples or Rules? Attributes
READ> Filename? voyage.att
Reading attributes...
Finished reading attribute file!
CN> Read
READ> Both, Atts, Examples or Rules? Rules
READ> Filename? voyage.cn2.unordered-rules.out
Reading rules...
Finished reading rules!
CN> Read
READ> Both, Atts, Examples or Rules? Examples
READ> Filename? voyage.tst1
Reading examples...
1 examples!
Finished reading example file!
CN> Execute
EVAL> all
Executing rules...
        PREDICTED
ACTUAL     go      dont_go Accuracy
      go     0       0    ---
 dont_go     1       0       0.0 %
Overall accuracy:   0.0 %
Default accuracy:   0.0 %

EVAL>
CN> Read
READ> Both, Atts, Examples or Rules? Examples
READ> Filename? voyage.tst2
Reading examples...
1 examples!
Finished reading example file!
CN> Execute
EVAL> all
Executing rules...
        PREDICTED
ACTUAL     go      dont_go Accuracy
      go     0       0    ---
 dont_go     0       1     100.0 %
Overall accuracy: 100.0 %
Default accuracy:   0.0 %
EVAL>
```

Figure 24: Consulting the voyage $\mathcal{CN}2$ unordered classifier

8.9 Summary

Never let formal education get in the way of your learning.

—Mark Twain

As can be seen, the inductive bias of $\mathcal{CN}2$ is quite different than $\mathcal{C}4.5$rules. In general, inducing rules is a more complex task than inducing decision trees or decision rules (which are extracted from decision trees). As in $\mathcal{C}4.5$, $\mathcal{CN}2$ implements two algorithms: one for inducing ordered rules (which can be seen as a degenerate binary tree) and one for inducing unordered rules.

Differently from decision trees, the rules generated by $\mathcal{CN}2$ are not disjunct, meaning that the same instance can be covered by different rules.

As can be observed, $\mathcal{CN}2$ does not allow to consult the extracted classifier with unlabeled instances. The only possible way to do that is creating one file containing one instance to be classified as well as a 'guessed' class label. After inducing the classifier (or reading the rule file) then it is necessary to load this one-instance file and to ask $\mathcal{CN}2$ to evaluating it using the `eXecute` option. If the predicted class is the same as the 'guessed' one, than the error rate will be zero. If not, the error rate will be non zero.

9 $\mathcal{R}$*ipper*

Anyone who has never made a mistake has never tried anything new.

—Albert Einstein

$\mathcal{R}$*ipper* — Repeated Incremental Pruning to Produce Error Reduction — is a program for inducing classification rules from a set of labeled instances (Cohen, 1995). An interesting point about $\mathcal{R}$*ipper* is that it allows the user to specify constraints on the format of the learned if-then rules. If there is some prior background knowledge about the concept to be learned, then these constraints can often lead to more accurate hypotheses.

Also, besides allowing nominal and continuous features, $\mathcal{R}$*ipper* deals with "set-valued" features. The value of a set-valued feature is a set of atoms: for example, a set-valued feature could be used to encode the set of words that appear in a document. Recent versions of $\mathcal{R}$*ipper* also support bag-valued[2] features.

Induced rules by $\mathcal{R}$*ipper* assumes the Prolog clausal notation:

$$C_i \;\; \textbf{:-} \; X_r \ \mathrm{op}_r \ \mathrm{Value}_r, \ X_s \ \mathrm{op}_s \ \mathrm{Value}_s, \ \ldots \ , \ X_t \ \mathrm{op}_t \ \mathrm{Value}_t \ (P/N)$$

which can be seen as

$$C_i \;\; \textbf{:-} \; \text{<complex>} \ (P/N)$$

where X_i is a feature, *op* is an operator in the set $\{=, \neq, <, \leq, >, \geq\}$ and *Value* is a valid feature X_i value. Commas separating each test stand for logical **and**. Numbers between parenthesis

[2] A bag is a data structure similar to a set but allowing that items in the bag appear more than once.

Algorithm 4 IREP

```
 1: procedure IREP(Pos,Neg)
 2: Ruleset := {}
 3: while Pos ≠ {} do
 4:    Split (Pos,Neg) into (GrowPos,GrowNeg) and (PrunePos,PruneNeg)
 5:    Rule := GrowRule(GrowPos,GrowNeg)
 6:    Rule := PruneRule(Rule,PrunePos,PruneNeg)
 7:    if the error rate of Rule on (PrunePos,PruneNeg) exceeds 50% then
 8:       return Ruleset
 9:    else
10:       Ruleset := Ruleset + {Rule}
11:       remove instances covered by Rule from (Pos,Neg)
12:    end if
13: end while
14: return Ruleset
```

have the following meanings: P indicates the number of covered instances that satisfies rule conditions and belongs to class C_i; N is the number of covered instances that satisfies rule conditions and does not belong to class C_i.

9.1 Inducing Rules

After arranging the classes (see `-a` *Ordering* option) $\mathcal{R}ipper$ tries to find a rule set that separates class C_1 from classes $\{C_2, C_3, \ldots, C_k\}$. After that, all instances covered by the learned rule set (including true and false positives for class C_1) are removed from the training set and $\mathcal{R}ipper$ repeats this process in order to separate class C_2 from classes $\{C_3, C_4, \ldots, C_k\}$ and the cycle is repeated.

The final class C_k will become the default class. The final result is that rules for a single class will always be grouped together, but rules for class C_i are possibly simplified, because they can assume that the class of the instance is one of $\{C_i, C_{i+1}, \ldots, C_k\}$.

9.2 Handling Continuous Values

$\mathcal{R}ipper$ is based on an improved version of IREP — Incremental Reduced Error Pruning — originally proposed by (Fürnkranz and Widmer, 1994) which supports only two-class problems. The improved version is known as IREP* and incorporates modifications proposed in (Cohen, 1995). The IREP* rule learning algorithm supports unknown values, numerical features and multiple classes. Algorithm 4 presents a two-class version of IREP. Algorithm 5 shows the $\mathcal{R}ipper$ inducer as well as its associated Algorithm 6 for optimizing the ruleset and Algorithm 7 for building the ruleset (Cohen, 1995; Dietterich, 1997).

9.3 Handling Unknown Values

Missing values are treated in a manner such that all tests involving unknown values for a given feature X_i are defined to fail on those instances for which the value of X_i is missing. This is done

Algorithm 5 $\mathcal{R}ipper$

```
1: procedure Ripper(P, N, k)
2: RuleSet := BuildRuleSet(P, N)
3: for i := 1 to k do
4:    RuleSet := OptimizeRuleSet(RuleSet, P, N)
5: end for
6: return Ruleset
```

to force the inducer to separate out instances that belong to class C_i from $\{C_{i+1}, C_{i+2}, \ldots, C_k\}$ using tests that are known to succeed.

9.4 Handling Noisy Instances

The IREP algorithm can handle, in a limited way, noisy instances by pruning only a single final rule condition. On the other hand, IREP* can handle efficiently noisy instances since it can prune any number of final rule conditions.

9.5 Classifying New Instances

Given a new instance to $\mathcal{R}ipper$, each rule is tried until one fires. If an instance is covered by rules from two or more classes, then the first rule to fire classifies the new instance.

Algorithm 6 $\mathcal{R}ipper$ Optimize Rule Set

```
1: procedure OptimizeRuleSet(RuleSet, P, N)
2: for all rule R ∈ RuleSet do
3:    RuleSet := RuleSet − {R}
4:    UPos := instances in P not covered by RuleSet
5:    UNeg := instances in N not covered by RuleSet
6:    split (UPos, UNeg) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
7:    RepRule := GrowRule(GrowPos, GrowNeg)
8:    RepRule := PruneRule(RepRule, PrunePos, PruneNeg)
9:    RevRule := GrowRule(GrowPos, GrowNeg, R)
10:   RevRule := PruneRule(RevRule, PrunePos, PruneNeg)
11:   BetterRule := choose better of RepRule and RevRule
12:   RuleSet := Ruleset + {BetterRule}
13: end for
14: return Ruleset
```

9.6 Program Names

ripper This command invokes rule learning (see Section 9.8 for options of this command).

boost Learn a ruleset from instances using boosting.

predict Predict classes of new instances using a ruleset. Output is a 'prediction file' in which each line corresponds to an instance and contains:

predicted-class P N true-class.

Algorithm 7 $\mathcal{R}ipper$ Build Rule Set

```
1: procedure BuildRuleSet(P, N)
2: P := positive instances
3: N := negative instances
4: RuleSet := { }
5: DL := DescriptionLength(RuleSet, P, N)
6: while P ≠ { } do
7:    // Grow and prune a new rule
8:    split (P, N) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
9:    Rule := GrowRule(GrowPos, GrowNeg)
10:   Rule := PruneRule(Rule, PrunePos, PruneNeg)
11:   RuleSet := RuleSet + {Rule}
12:   if DescriptionLength(RuleSet, P, N) > DL + 64 then
13:      // Prune the whole rule set and exit
14:      for all rule R ∈ RuleSet (considered in reverse order) do
15:         if DescriptionLength(RuleSet − {R}, P, N) < DL then
16:            RuleSet := RuleSet − {Rule}
17:            DL := DescriptionLength(RuleSet, P, N)
18:         end if
19:      end for
20:      return Ruleset
21:   end if
22:   DL := DescriptionLength(RuleSet, P, N)
23:   P := P − all instances covered by Rule
24:   N := N − all instances covered by Rule
25: end while
26: return Ruleset
```

corrupt Add class noise to data.

summarize Summarize a prediction file, such as total number of examples, error rate, error rate in percent, etc.

verify Do a consistency check on a dataset.

partition Choose a random subset of a dataset.

filter-text Remove words from a set-valued fields in a dataset.

phrases Add phrases to a set-valued fields of a dataset.

select-class Change class of examples of anything other than a given class to the default class.

rocchio Use Rocchio's method to learn a classifier.

pair Perform a paired test on two prediction files.

eliminate Eliminate redundant conditions from a ruleset.

test-rules Test the ruleset `.hyp` on the `.test` file and change the counts associated with each rule.

pprint-rules Print a ripper ruleset.

clean-data Clean a dataset by removing instances with inconsistent labels.

transform Transform a dataset from the standard representation to an indication of which rules fired for which examples using rules from a given set of hypotheses.

data2text Convert ripper dataset (`.data` and `.test` files) to text.

9.7 Files

All files read and written by *Ripper* assume the form *filestem.extension*, where *extension* characterizes the type of data involved. *filestem* is the first and only argument to *Ripper*. Anything following a percent sign character until the end of line is a comment.

`.names` This file defines the names of the classes and features used in the `.data` file. If there is no names file, *Ripper* will assign arbitrary names to the features and classes, and will try to figure out the types of the features from the data.

The `.names` file contains first a list of atoms representing the classes separated by commas and terminated by a period. An atom contains only letters, digits, and the underscore character, and must begin with a letter. Alternatively, an atom is any sequence of characters enclosed in single quotes. The list of classes is followed by a list of feature definitions. Each feature definition consists of the name of the feature and a colon followed by:

- continuous if the feature is continuous;
- set if the feature is set-valued;
- bag if the feature is bag-valued;
- symbolic if the feature can take any symbolic value;
- ignore if *Ripper* should completely ignore the feature;
- suppressed if the feature should be suppressed. This feature type is similar to ignore, except that while it is not used in *Ripper*'s hypotheses, the number of values of the feature does effect MDL-based pruning. Hence, suppressing a feature that was not used in a hypothesis should not change *Ripper*'s performance in any way;
- a comma-separated list of atoms representing possible symbolic values of the feature, if the feature is nominal.

Finally, every feature definition must be terminated by a period.

`.data` This file contains the data with some preclassified instances. An instance for *Ripper* is described by a fixed set of features. In the data file each instance is a comma-separated list of feature values, followed by an atom indicating the class of the instance, followed by a period.

Feature values, separated by commas, are given in the same order that features are defined in the names file; most of the usual syntax for numbers are supported. Set- and bag-valued features are specified by simply enumerating the elements of the set, separated with white space. Unknown features are indicated with a question mark (?).

Instances can also be given a weight, by inserting `:`w between the class name and the terminating period, where w is a real number. The default value is $w = 1$.

`.test` The `.test` file contains some additional preclassified instances to be used as test cases. The test file is formatted in the same way as the data file. If there is no test file, $\mathcal{R}$*ipper* will either not test the learned rule set or, if the user chooses the `k` or `y` options, $\mathcal{R}$*ipper* will use cross-validation to test the learned rule set. As can be seen, the format for these three previous files is roughly the same as the one used by $\mathcal{C}4.5$.

`.hyp` This file contains the learned rules.

`.gram` The `.gram` file contains a description of a grammar in BNF notation. The grammar file is optional for $\mathcal{R}$*ipper*, and most users will probably not want to change the default grammar described below

This file contains a grammar defining the rules that are allowed to be used in a hypothesis. The terminal symbols of the grammar are tests on the values of features defined in the `.names` file. Each sentence generated by the grammar is thus a sequence of feature value tests. $\mathcal{R}$*ipper* will read in this grammar and constrain its learning so that every rule generated by will have as an antecedent a sequence of feature value tests that is a sentence of the grammar. Thus, the grammar is a way for the user to guide $\mathcal{R}$*ipper*'s choice of rules.

More specifically, the grammar file contains a series of grammar rules. Each grammar rule consists of an atomic left–hand side followed by `-->` followed by a comma-separated list of grammar symbols followed by a period. A grammar symbol is either a nonterminal symbol (which is simply an atom that appears on the left-hand side of some grammar rule) or a terminal symbol. A terminal symbol is of the form

$$X_i \ op \ Value$$

where X_i is the name of a feature and *Value* is a valid value for that feature. The operator *op* must be one of $\{=, != , >=, <=, \sim, !\}$. Terminal symbols of the form

$$X_i \ op \ *$$

are also allowed, in which case any possible value, indicated by '*', is legal. In general, $\{\sim, ! \sim\}$ are used in the following situations:

- $X_i \sim Value$ is used for set- and bag-valued features. This condition is true if X_i is a set-valued and *Value* is contained in the set. For bags, the condition $X_i \sim Value_w$ is true if feature contains at least w instances of *Value*.
- $X_i \, !\!\sim Value$ is also used for set- and bag-valued features. This condition is true if *Value* is not present in the set.

Finally, prefixing a grammar rule with an exclamation point indicates to $\mathcal{R}$*ipper* that sentences generated using that grammar rule have a lower priority; if possible, $\mathcal{R}$*ipper* will build a hypothesis without using low priority sentences. Even lower priorities can be assigned by prefixing grammar rules with a string of two or more exclamation points.

When learning rules to predict the class C_i, $\mathcal{R}$*ipper* will expect to find some left-hand side of the form "body_C_i" to use as the start symbol of the grammar; if this is not present, $\mathcal{R}$*ipper* will use the atom "body" as the start symbol. If this is not present, $\mathcal{R}$*ipper* will construct the following default grammar:

```
body --> body_conds.
body_conds --> .
body_conds --> cond, body_conds.
cond --> feature1_cond.
⋮
cond --> featurek_cond.
```

where `feature1, ... , featurek` are the names of the features defined in the `.names` file. If discretization is used, then for each continuous feature `cfeature`, the default grammar also contains the rules

```
cfeature_cond --> cfeature >= t1.
cfeature_cond --> cfeature <= t1.
⋮
cfeature_cond --> cfeature >= tn.
cfeature_cond --> cfeature <= tn.
```

where `t1, ... , tn` are *Ripper*'s discretization of the training data. Otherwise, the grammar will contain the rules

```
cfeature_cond --> cfeature >= '*'.
cfeature_cond --> cfeature <= '*'.
```

For a nominal feature `nfeature` the default grammar contains the rule

```
nfeature_cond --> nfeature = '*'.
```

For a set- or bag-valued feature `sfeature` the default grammar contains the rules

```
sfeature_cond --> sfeature ~ '*'.
sfeature_cond --> sfeature !~ '*'.
```

If the grammar file is missing or empty, then the default grammar will be used. If the grammar contains definitions of some but not all of the nonterminal symbols used in the default grammar, they will override the default definitions.

9.8 *Ripper* Options

For running *Ripper*, the command line assumes the form:

ripper [*options*] *filestem*

Options of *Ripper* have the following meanings:

- `-c`
  Inform *Ripper* that noise-free data is expected.

- `-n`
  Inform *Ripper* that noisy data is expected. This is the default.

- `-k`v
  Define the number of partitions for cross–validation as v. There is no default value.

- `-l`
  Define leave-one-out for cross-validation *i.e.*, n-fold cross-validation where n is the size of training set.

- `-v`L
  Set the verbosity level to L, which can take values 0, 1, 2, or 3. The default is 0.

- `-a`*Ordering*
  Before learning, *Ripper* first heuristically orders the classes by one of the following methods:

  - +freq, order by increasing frequency (default);
  - −freq, order by decreasing frequency;
  - given, order classes as in the `.names` file;
  - mdl, use heuristics to guess an optimal ordering;
  - unordered.

  With the '`-a`*Ordering*' option, *Ripper* will separate each class from the remaining classes, thus ending up with rules for every class. Conflicts are resolved by deciding in favor of the rule with lowest training set error.

- `-s`
  Indicate that the training set should be read from standard input, rather than from *filestem*`.data`.

- `-g` *filestem*
  Indicate that the grammar file *filestem*`.gram` should be used.

- `-f` *filestem*
  Indicate that the names file *filestem*`.names` should be used.

- `-O`v
  Control optimization of rules. *Ripper* makes v optimization passes over the rules it learns. The default is $v = 2$. For $v = 2$, *Ripper* is named *Ripper* 2 and for $v = k$, *Ripper* k.

- `-M`v
  Indicate the use of statistics collected on a class-stratified subsample of v instances (instead of the entire dataset) to make certain frequently repeated decisions. For very large datasets, *Ripper* using subsamples of a few hundred or a few thousand will typically produce a slightly inferior ruleset; however, it will run much more quickly than *Ripper* without subsampling.

- `-I`v
  Indicate to discretize continuous features into v equal frequency segments. If $v = 0$, it discretizes into the maximal possible number of segments. Default is to not discretize continuous values. Discretization usually speeds up *Ripper* on large datasets with many continuous values, but may cost in accuracy.

- `-G`
  Print the grammar and exit. It is useful to make a change to the default grammar.

- `-N`
  Tell *Ripper* to print the `.names` file and exit. This is sometimes useful to generate a `.names` file to be used by *C*4.5. *Ripper* can usually infer the types of a feature from a dataset, so a `.names` file for *Ripper* is optional.

- `-R`
  Allow to randomize *Ripper*'s operation. By default, a fixed random seed is used.

- `-t`*string*
  Allow or disallow negative tests in rules. If the *string* contains a "s", then negative tests of the form "X_i !∼ Value" for set- and bag-valued features will be allowed in rules. The symbol !∼ stands for "does not contain". If the string contains an "n", then negative tests of the form "X_i != Value" for nominal features will be allowed in rules.

- `-D`v
  Change the maximum decompression.

- `-S`v
  Simplify the hypothesis more ($v > 1$) or less ($v < 1$).

- `-L`v
  Change the loss ratio *i.e.*, the ratio of the cost of a false negative to the cost of a false positive. A value of $v > 1$ will usually improve recall of the minority classes, and a value of $v < 1$ will usually improve precision.

- `-A`
  Add redundant tests to rules. This sometimes improves precision and readability, mainly for set- or bag-valued features.

- `-F`v
  Force each rule to cover at least v instances.

9.9 Example

In what follows the voyage data (Section 4.10, page 10) will be used to show a simple *Ripper* execution.

9.9.1 Input Data

Figures 25 and 26 show the corresponding `voyage.names` and `voyage.data`, respectively. For this example, no modifications where performed in the standard `voyage.gram` file (the default one was used instead).

9.9.2 The Classifier

Running *Ripper* with default options in the voyage example gives just the default rule for class **go**. So, we have used the `-c` option to inform *Ripper* that data is noisy free:

```
go, dont_go.        % Classes

% Features
outlook:     sunny, overcast, rain.
temperature: continuous.
humidity:    continuous.
windy:       yes, no.
```

Figure 25: The *Ripper* `voyage.names` file

```
sunny,   25,72,yes,go.
sunny,   28,91,yes,dont_go.
sunny,   22,70,no, go.
sunny,   23,95,no, dont_go.
sunny,   30,85,no, dont_go.
overcast,23,90,yes,go.
overcast,29,78,no, go.
overcast,19,65,yes,go.
overcast,26,75,no, go.
overcast,20,87,yes,dont_go.
rain,    22,95,no, go.
rain,    19,70,yes,dont_go.
rain,    23,80,yes,dont_go.
rain,    25,81,no, go.
rain,    21,80,no, go.
```

Figure 26: The *Ripper* `voyage.data` file

ripper -c voyage

Figure 27 shows the output generated by $\mathcal{R}ipper$. Figure 28 shows the grammar produced by $\mathcal{R}ipper$ using the -G option.

```
option: data is clean
Final hypothesis is:
dont_go :- humidity>=85, outlook=sunny (3/0).
dont_go :- windy=yes, outlook=rain (2/0).
dont_go :- temperature<=20, temperature>=20 (1/0).
default go (9/0).
============================== summary ==============================
Train error rate:  0.00% +/- 0.00% (15 datapoints)    <<
Hypothesis size:   3 rules, 9 conditions
Learning time:     0.01 sec
```

Figure 27: The voyage $\mathcal{R}ipper$ classifier

9.9.3 Consulting the Classifier

The same two test instances used for $\mathcal{C}4.5$ will be used here *i.e.*, (outlook, temperature, humidity, windy) = (sunny, 35, 70, no) and (?, 30, 80, yes) — see Section 4.10.2, page 12.

For labeling data, both test instances were placed into the voyage.test file with a "guessed" class label (dont_go) in this form:

```
sunny,35,70,no ,dont_go.
?,    30,80,yes,dont_go.
```

and the following command was used:

predict voyage

The output of this command assumes the form predicted-class P N true-class (see page 51) and it is shown in Figure 29 where, for both instances, the predicted class label is go.

```
body --> body_conds.
body_conds --> .
body_conds --> cond,body_conds.
cond --> cond_outlook.
cond --> cond_temperature.
cond --> cond_humidity.
cond --> cond_windy.
cond_humidity --> humidity<='*'.
cond_humidity --> humidity>='*'.
cond_outlook --> outlook='*'.
cond_temperature --> temperature<='*'.
cond_temperature --> temperature>='*'.
cond_windy --> windy='*'.
```

Figure 28: The voyage grammar

```
go 9 0 dont_go go 9 0 dont_go
```

Figure 29: Consulting the *Ripper* voyage rule classifier

9.10 URL

You can find more information about *Ripper* in the URL
`http://www.research.att.com/~wcohen/ripperd.html`

9.11 Summary

> Every revolutionary idea seems to evoke three stages of reaction. They may be summed up by the phrases: (1) It's completely impossible. (2) It's possible, but it's not worth doing. (3) I said it was a good idea all along.
>
> —*Arthur C. Clarke*

The basic strategy used by RIPPER k to find a rule set that models the data is to first use IREP* to find an initial model, and then to iteratively improve that model, using the "optimization" procedure described in (Cohen, 1995). According to Cohen, this process seems to be more efficient because (1) building the initial model is efficient, (2) the initial model does not tend to be large relative to the target concept, and (3) the optimization steps only require time linear in the number of examples and the size of the initial model.

*C*4.5rules also constructs an initial model and then iteratively improves it. However, for *C*4.5rules, the initial model is a subset of rules extracted from an unpruned decision tree, and the improvement process greedily deletes or adds single rules in an effort to reduce description length. *C*4.5rules repeats this process for several different-sized subsets of the total pool of extracted rules and uses the best ruleset found as its hypothesis; the subsets it uses are the empty ruleset, the complete ruleset, and randomly chosen subsets of 10%, 20%, ... , and 90% of the rules.

However, for noisy datasets, the number of rules extracted from the unpruned decision tree grows as n, the number of instances. This means that each initial model will also be of size proportional to n, and hence if n is sufficiently large, all of the initial models will be much larger than the target hypothesis. This means that to build a theory about the same size as the target concept always requires many ($O(n)$) changes to the initial model, and at each step in the optimization, many (also $O(n)$) changes are possible. The improvement process is thus expensive; since it is a greedy search, it is also potentially quite unlikely to find the best ruleset.

Cohen also claims that *Ripper* is asymptotically faster than other competitive rule learning algorithms; this means that it will be much faster on large datasets and especially on large noisy datasets. Many existing systems like *C*4.5rules use pruning algorithms that are asymptotically cubic or even worse. *Ripper* is asymptotically $O(n \log^2(n))$.

The rules produced by RIPPER tend to be compact and understandable. On the test suite performed in (Cohen, 1995) *Ripper*'s rule sets were always smaller than *C*4.5rules — often by a factor of two or more.

10 Inducers Summary

The things which hurt, instruct.

—*Benjamin Franklin*

Table 2 shows a summary of some characteristics of the inducers discussed in this work. In this table n is the number of training instances; m is the number of features; s is the star size (for the $\mathcal{CN}2$ inducer); DT stands for "Decision Tree"; RI stands for "Rule Induction" and RDT stands for "Rules extracted from DT".

Complexity analysis for each inducer were obtained for $\mathcal{C}4.5$ and $\mathcal{C}5.0$ from (Murthy et al., 1994) assuming the worst-case running time for inducing axis–parallel trees (in fact this is a general analysis for decision trees). In (Cohen, 1995) the complexity analysis for $\mathcal{R}ipper$ can be found as well as some evidences about the cubic behaviour of $\mathcal{C}4.5$rules (which does not include the time for generating the tree), although in a personal communication Prof. Quinlan has ensured that 'there are no simple time complexities, either worst-case or average-case, for $\mathcal{C}4.5$ or $\mathcal{C}5.0$'. The complexity analysis for $\mathcal{CN}2$ can be found in (Clark and Niblett, 1989).

Characteristic	$\mathcal{C}4.5$	$\mathcal{C}4.5$rules	$\mathcal{C}5.0$	$\mathcal{OC}1$	$\mathcal{CN}2$	$\mathcal{R}ipper$
classifier type	DT	RDT	DT/RDT	DT	RI	RI
current release	8	8	1.11		5.1	
symbolic unordered features	yes	yes	yes		yes	yes
symbolic ordered features			yes			
integer features	yes	yes	yes	yes	yes	yes
real features	yes	yes	yes	yes	yes	yes
set-valued features						yes
bag-valued features						yes
date features			yes			
user defined features			yes			
ignore features	yes	yes	yes			yes
unknown value	?	?	?		?	?
don't care value					!	
weight instance					yes	yes
names file	yes	yes	yes		yes	yes
data file	yes	yes	yes	yes	yes	yes
test file	yes	yes	yes	yes	yes	yes
cost file			yes			
grammar file						yes
ordered rules		yes			yes	yes
unordered rules	yes		yes	yes	yes	
complexity	$O(mn^2)$	$O(n^3)$	$O(mn^2)$	$O(mn^2 \log(n))$	$O(m^2n^2s)$	$O(n \log^2(n))$

Table 2: Inducers Summary

11 Concluding Remarks

The best way to have a good idea is to have a lot of ideas.

—*Linus Pauling*

Machine Learning is a powerful tool for overcoming the bottleneck of knowledge acquisition, and several algorithms have already been implemented and new ones will be implemented in the

future. The very basic (but still important) lesson to be learnt is that there is no algorithm that dominates all others for all problems. The best we can hope for is to understand the strengths and limitations of different algorithms, and based on background knowledge for a given domain, make recommendations as to which algorithms to use.

In this work we discuss the facilities provided by six well known inducers which express concept description in the form of decision trees and decision rules: $\mathcal{C}4.5$ and $\mathcal{OC}1$ for decision trees and $\mathcal{C}4.5$rules, $\mathcal{CN}2$ and $\mathcal{R}ipper$ for rules as well as $\mathcal{C}5.0$ for decision trees and rules. Our main objective is to describe those inducers under a common framework, such that the main differences as well as similarities among them can easily be observed. For those six inducer we also present a thorough description of the necessary commands for executing each inducer, files used, list and meaning of all possible command line options as well as an example of execution. As this sort of information, although using different notations, is spread out in several papers and manuals, we expect to facilitate the initial work of users that wish to experiment with those inducers.

Acknowledgments: We are grateful to Huei Diana Lee and Jaqueline Brigladori Pugliesi for helpful comments on a draft of this report.

References

Baranauskas, J. A. and Monard, M. C. (2000). Reviewing some machine learning concepts and methods. Technical Report 102, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_102.ps.zip.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees*. Wadsworth & Books, Pacific Grove, CA.

Clark, P. and Boswell, R. (1991). Rule induction with $\mathcal{CN}2$: Some recent improvements. In Kodratoff, Y., editor, *Proceedings of the 5th European Conference (EWSL 91)*, pages 151–163. Springer-Verlag.

Clark, P. and Niblett, T. (1987). Induction in noise domains. In Bratko, I. and Lavrač, N., editors, *Proceedings of the Second European Working Session on Learning*, pages 11–30, Wilmslow, UK. Sigma.

Clark, P. and Niblett, T. (1989). The $\mathcal{CN}2$ induction algorithm. *Machine Learning*, 3(4):261–283.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, San Francisco, CA. Morgan Kaufmann.

Dietterich, T. G. (1997). Machine learning research: Four current directions. ftp://ftp.cs.orst.edu/pub/tgd/papers.

Fürnkranz, J. and Widmer, G. (1994). Incremental reduced error pruning. In Cohen, W. W. and Hirsh, H., editors, *Proceedings of the Eleventh International Conference on Machine Learning*, pages 70–77, New Brunswick, New Jersey. Morgan Kaufmann.

Murthy, S. K., Kasif, S., and Salzberg, S. L. (1994). A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2(1):1–32. http://www.cs.jhu.edu/~salzberg/jair94.ps.

Prati, R. C., Baranauskas, J. A., and Monard, M. C. (1999). BibView: Um sistema para auxiliar a manutenção de registros para o BibTeX. Technical Report 95, ICMC-USP.

ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_95.ps.zip.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106. Reprinted in Shavlik and Dieterich (eds.) Readings in Machine Learning.

Quinlan, J. R. (1987a). Generating production rules from decision trees. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 304–307, Italy.

Quinlan, J. R. (1987b). Simplifying decision trees. *International Journal of Man-Machine Studies*, 27:221–234.

Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning*. Morgan Kaufmann, CA.

Salzberg, S. L. (1995). Locating protein coding regions in human dna using a decision tree algorithm. *Journal of Computational Biology*, 2(3):473–485.

Salzberg, S. L., Chandar, R., Ford, H., Murthy, S., and White, R. (1995). Decision trees for automated identification of cosmic ray hits in Hubble space telescope images. *Publications of the Astronomical Society of the Pacific*, 107:1–10.

Weiss, S. M. and Indurkhya, N. (1998). *Predictive Data Mining: A Practical Guide*. Morgan Kaufmann, San Francisco, CA.