Abstract

The modular nature of rules learned by most inductive machine learning algorithms makes them difficult and costly to maintain when the knowledge they are based on changes. Ripple-down rules, a tree-like rule-based concept description format, overcomes these problems by using rules that operate in the context of other rules.

Whereas most algorithms use the standard propositional attribute–value representation for input, relational learning with examples expressed in first-order languages opens up a myriad of new classification tasks, and new approaches to existing tasks. However, current relational learning algorithms employ simple rule formats that lack the desirable features of ripple-down rules.

This thesis describes a system that combines techniques of propositional and relational learning, and generates a concept description in ripple-down rule form. The system can learn from purely propositional data, employing both propositional and relational techniques, or relational data, using background knowledge presented as extensional relation definitions. The system illustrates several benefits of the combined approach, and its performance is compared with existing propositional and relational learners, such as C4.5 and FOIL.

Contents

1. Introduction..
1.1 More expressive concept descriptions..
1.2 A different approach to rules..
1.3 HENRY..
1.4 Thesis outline..
2. Ripple-down Rules..
2.1 Ripple-down rule structure..
Concept descriptions using ripple-down rules..
Rule structure..
2.2 INDUCT..
Approaches to separating classes..
PRISM..
Statistically well-founded induction..
Induction of ripple-down rules..
2.3 Nested Generalised Exemplars..
Exceptions in exemplar-based learning..
Comparing nested generalised exemplars and ripple-down rules..
3. Relational Rules..
3.1 Learning relations..
Standard relations..
User relations..
Representing relations..
3.2 Inductive logic programming..
Logic programming..
Database terminology..
Empirical inductive logic programming..
3.3 FOIL..
Defining the search space..
Restricting the search space..
Evaluating literals..
Learning the concept of an arch..
4. HENRY and ABE..
4.1 HENRY..
Implementation overview..

Learning propositional terms
Learning relations
Variables
4.2 Ripple-down rule output formats
Text-based format
Lisp format
Dotty format
Translating ripple-down rules to PROLOG
4.3 ABE
Input and output
Matching examples
Missing values
5. Results
5.1 Experiments on UCI data sets
Comparing propositional algorithms
Comparing rule evaluation heuristics
Comparing relational algorithms
Comparing missing value methods
5.2 Experiments on relational data sets
The arches problem
The trains problem
The text classification problem
The numbers problem
6. Conclusions
Combining propositional and relational terms
Experimental results
Negated literals
Ripple-down rules
Missing values
References
Appendix
The UCI datasets
The *arches* dataset
arches.arff
arch_1rels.arff
arch_2rels.arff
arches.foil

The *trains* dataset..

trains.arff...

train_1rel.arff...

trains_2rel.arff...

trains.foil..

The *numbers* data...

nums3.foil...

nums3.arff...

nums3_1rel.arff...

nums3_2rel.arff...

Figures

Figure 2.1 A disjunctive normal form rule set describing the iris concept that might be used in an expert system..
Figure 2.2 Part of a ripple-down rule structure used in a medical diagnosis system (from Gaines and Compton, 1995)..
Figure 2.3 Classification of an example that fails to match rules in a ripple-down rule tree..
Figure 2.4 A two-dimensional instance space containing examples of two classes, *n* and *p*. ....
Figure 2.5 The instance space of Figure 2.4 divided by the term $x > 0.1$............................
Figure 2.6 The instance space of Figure 2.5 further divided by the term $y > 12$.................
Figure 2.7 The decision tree in Figure 2.7 with the right subtree divided, this time by the attribute *y*..
Figure 2.8 A decision tree that divides the data from Figure 2.4 into two subsets................
Figure 2.9 Pseudocode outline of the PRISM covering algorithm.....................................
Figure 2.10 Pseudocode outline of the best_clause function used by INDUCT...................
Figure 2.11 Curve showing how local minima in the *m*-function may be encountered as a rule is generated..
Figure 2.12 A set theory representation of INDUCT's probabilistic rule evaluation heuristic..
Figure 2.13 Pseudocode outline of the INDUCT(RDR) algorithm......................................
Figure 2.14 Two example instance spaces divided by DNF rule sets..................................
Figure 2.15 An instance space divided into regions predicting four classes.........................
Figure 2.16 The instance space in Figure 2.15 showing the effect of moving the right-hand bound on the region predicting class *p*..
Figure 2.17 The instance space of Figure 2.15 with an exception region surrounding the new example of class *r*..
Figure 2.18 A Euclidean distance function used to determine the stored example most similar to the new example..
Figure 2.19 An example two-dimensional instance space with two hyper-rectangular exemplars and one point exemplar..
Figure 2.20 The new example is classified as a *p* because it falls within the hyper-rectangle..
Figure 2.21 Instance-based and generalised-exemplar methods for classifying a new example..

Figure 2.22 Examples falling near the point exemplar are classified as class *p* because they fall inside the hyper-rectangle..

Figure 2.23 Two point exemplars in the class *p* hyper-rectangle are joined forming an exception exemplar..

Figure 3.1 Eight blocks representing the "standing up" concept.......................................

Figure 3.2 An ARFF file containing information about the width, height, and number of edges of the building blocks in Figure 3.1..

Figure 3.3 Two propositional rules describing the "standing up" concept generated from the data file in Figure 3.2..

Figure 3.4 A block misclassified by the rules in Figure 3.3..

Figure 3.5 Two rules that describe "standing up" concept that specify relationships between the attributes of the blocks...

Figure 3.6 An ARFF file representing technical report abstracts as word-appearance tuples..

Figure 3.7 Two propositional rules that define positive examples from the set of abstracts..

Figure 3.8 An ARFF file representing the abstracts using word positions as attributes.........

Figure 3.9 Extensional definitions for relations used in the text categorisation example....

Figure 3.10 Two relational rules that define positive examples from the set of abstracts....

Figure 3.11 A FOIL input file representing the blocks in the "standing up" problem..........

Figure 3.12 Extensional predicate definitions describing the positions of words in the technical report abstracts..

Figure 3.13 An intentional definition for the relation *after*...

Figure 3.14 Derivation of the truth value of after(word#2,word#4) from the intentional definition in Figure 3.13..

Figure 3.15 Examples of valid variables, and predicate and function symbols....................

Figure 3.16 Examples of valid terms and constants..

Figure 3.17 Intentional and extensional predicate definitions providing background knowledge for a relational learning task..

Figure 3.18 The set of ground facts derived from the predicate definitions in Figure 3.17..

Figure 3.19 Part of a FOIL input file for the "arches" problem..

Figure 3.20 Four examples used to learn the "arch" concept..

Figure 3.21 Example FOIL output for the "arches" problem...

Figure 4.1 Pseudocode of HENRY's best_clause function...

Figure 4.2 Example output for HENRY using relational terms on the "standing up" data....

Figure 4.3 Pseudocode outline of best_clause used in the relational extension to HENRY....

Figure 4.4 Pseudocode outline of the best_clause function used to search the space of user-defined relations..

Figure 4.5 Text-based ripple-down rule tree generated from the iris data...........................

Figure 4.6 Lisp-based ripple-down rule tree generated from the iris data............................
Figure 4.7 Ripple-down rules for the iris data as displayed by DOTTY..............................
Figure 4.8 DOTTY input file for the ripple-down rules generated from the iris data...........
Figure 4.9 Translation of ripple-down rules to an equivalent PROLOG-like format...........
Figure 4.10 Output from ABE evaluating ripple-down rules on the iris data.......................
Figure 5.1 A section of a relational ripple-down rule tree generated from the IR data set...
Figure 5.2 Relational ripple-down rules generated by HENRY for the arches problem..........
Figure 5.3 Relational ripple-down rules generated by HENRY for *westbound* trains.............
Figure 5.4 Relational ripple-down rules generated by HENRY for *eastbound* trains.............
Figure 5.5 Relational ripple-down rules generated by HENRY for the text classification problem..
Figure 5.6 Relational ripple-down rules generated by HENRY for the numbers problem.......

Tables

Table 3.1 Relating database and logic programming terms (from Lavrac and Dzeroski, 1994)..........

Table 4.1HENRY's command line parameters..........

Table 5.1 Accuracy results for eight learning schemes on sixteen UCI data sets using the method described by Holte (1993)..........

Table 5.2 Accuracy results for two missing value evaluation methods on seven data sets with more than 5% missing values..........

1. Introduction

Machine learning algorithms examine a set of examples of a concept and generate a description of the concept that summarises the training data, and can be used to predict whether or not unseen examples also belong to the concept. A key issue is the way in which the concept description is expressed. Two common methods are as an explicitly stated generalisation, or a set of representative examples stored verbatim. The two forms result from different methods of learning, and require different methods of evaluation. Each is suited to a different set of learning tasks that are distinguishable by the accessibility of training data, and the expected use of the concept description.

Explicit generalisations are usually produced from a fixed set of training examples, and evaluated by matching terms in the concept description with the corresponding features in the examples. When learning such a description, most of the computation involves discovering regularities in the training data that lead to the generalisations. In contrast, algorithms that keep a set of representative examples defer the computational effort to the evaluation stage. These algorithms usually learn one example at a time, comparing each with examples already stored and storing it also if it is sufficiently different. In this approach there is no difference between learning and classification of new examples, because all training examples are themselves classified like new examples at some point. New examples are predicted to have the same class as the stored example or examples that are most similar with respect to all the features of the objects.

Both methods of creating concept descriptions have been shown to be effective in many classification tasks, with neither method having a significant advantage. However, algorithms that produce explicit concept descriptions are favoured in machine learning applications because the concept description can be used if the knowledge implicit in the examples is required. Human experts can examine the concept description to identify important features of the concept, and to provide explanations of classifications. Conversely, algorithms using the exemplar-based approach tend to be “black boxes” issuing a classification for an example with the only explanation being “it was closest to an example of that class.” Although different measures of similarity are used by different exemplar-based algorithms, they all essentially measure the distance between examples.

Inductive machine learning algorithms usually generate explicit concept descriptions in the form of a decision tree or set of production rules. The most common rule form is disjunctive normal form (DNF), although others have been used. Decision trees are readily converted into DNF form, and this is often necessary for manipulation by programs other than those that generated the trees. A rule set in disjunctive normal form consists of a

number of conjunctive clauses for each class of the concept. An example of the concept is classified by any rule that it matches. DNF rules are considered to be an intuitive form for expressing many concepts, and are used widely in knowledge-based systems.

1.1 More expressive concept descriptions

Many inductive learning algorithms use zero-order propositional languages to describe concepts. Examples are represented as tuples of values for a fixed set of attributes, and concept descriptions are expressed with propositional terms using these attributes. Typical examples of this type of term are *length* < 23.5 and *colour* = *red*. An example matches a term if its value for the term's attribute has the specified relationship with the term's value. For example, a green building block that is 15 mm long will match the first term above, but not the second.

Although propositional rules are adequate for many classification tasks, there are problems for which they are unsuited. In general, these problems are represented by examples that are expressed as sets of objects, and the concept descriptions require rules that specify relationships between the objects. For example, several instances used to learn the concept "stack" in a blocks world domain might be represented by three blocks each. Information about the colour and size of the individual blocks is not important to this concept, whereas positional relationships between the blocks making up each example are important. A propositional algorithm might produce a very specific concept description stating that a stack must contain block A, block B and block C, or it must contain block D, block E and block F, and so on. A relational algorithm, given some background knowledge about the positions of the individual blocks, can state that a stack must contain three blocks, with the first on top of the second and the second on the third. This same background information can be provided for the propositional learner by expressing every relation as an attribute that indicates its relevance to each example, but this representation is cumbersome, and the resulting concept description will still be specific to the values expressed in the data.

Most algorithms that learn with relational rules fall into the category of inductive logic programming, a discipline that uses the firm theoretical background of computational logic to formalise and constrain the learning process. The first-order languages used by inductive logic programming systems are rigourously defined and form a subset of the PROLOG logic programming language. The task of such systems can be viewed as inducing a PROLOG program which will return the value *true* when an example of the target concept is presented as a query, and *false* when the example is not of the target concept. Because of the computational nature of first-order clauses, algorithms must be careful to avoid problems such as infinite recursion and inefficient computation resulting from ill-chosen relational terms.

A logic program can also be viewed as a set of DNF rules. In fact, propositional DNF rules are easily converted to PROLOG although they do not conform to some of the restrictions of inductive logic programming. A common format such as PROLOG allows a unified approach to evaluating concept descriptions.

1.2 A different approach to rules

Despite their widespread use in both propositional and relational forms, DNF rules are difficult to maintain and update in changing information environments. Because each rule is executed in isolation, its classification is not affected by the other rules. Therefore, unseen examples that were not represented accurately by the training data may be classified by many or none of the rules in the set. Multiple classifications and non-classifications cause problems when evaluating rules and classifying new examples, because there is no satisfactory method for determining the correct classification in these situations.

Updating the rule set to provide a single correct classification for a new example is difficult and costly, because every rule must be examined to determine the effect of alterations on previously classified examples. An alternative approach is to provide exceptions to rules to cover misclassified examples. This means that only minimal changes need to be made to individual rules. There is no need to create a new rule to cover a misclassified example, or to alter an existing rule to exclude it. Instead, a child rule is added that distinguishes examples such as the new misclassified one from those of the correct class. Examples matching both the parent and child rule are given the child's classification instead of the parent's. Examples matching the parent rule only are given its classification.

If a concept description is created solely using exceptions to previous rules, a hierarchical structure results. Called a ripple-down rule, this has the form of a binary tree similar to the decision trees generated by ID3 (Quinlan, 1986) and CART (Breiman *et al*., 1984). The differences lie at the internal nodes. CART trees, like ripple-down rules, are binary, but they have only a single attribute–value relationship at each node. For example, a node might specify that all examples with a value less than 5 for the attribute *x* be classified by the left subtree, and all those with a value greater than 5 be classified by the right subtree. If the attribute is symbolic the node specifies an equality relationship instead of the inequality used for numeric attributes. Such a tree is a special case of a ripple-down rule tree where each rule is only a single term. In general, however, rules may have any number of terms.

ID3 trees are different again. Like CART trees they specify only a single attribute at each node. However, if it is symbolic, the node has a branch for each possible value of the attribute, so the tree need not be binary. Ripple-down rules have the advantage that the

rule at each node may be as specific (many terms) or as general (few terms) as is desired. Thus, a ripple-down rule tree that distinguishes two classes of a concept may have a single node where an ID3 or CART tree requires several.

The use of a tree-like structure has another advantage over the use of DNF rules. Often two classes will differ in only a few respects. A learning algorithm may have to use several of the classes' similarities to distinguish them from other classes. In a tree-like structure this may result in the two classes being distinguished by a node some way down the tree. A rule-based concept description, however, would duplicate large parts of rules in order to first differentiate examples from the other classes. The final few terms of these rules would distinguish the examples of the two classes in question.

1.3 HENRY

This dissertation describes the development of an inductive learning system that combines propositional and relational learning techniques, and generates a concept description in ripple-down rule form. The implementation, HENRY, is based on the INDUCT propositional ripple-down rule learner (Gaines, 1991; Gaines, 1995). In addition to the normal functionality of INDUCT, the new algorithm searches for relationships between attributes in data expressed in attribute–value form. However, HENRY is also able to use background knowledge in the form of extensionally defined relations to learn rules that contain relational terms.

HENRY is not an inductive logic programming system. Rather it is an extended propositional learner that makes use of relationships suspected to exist in the data. The user supplies HENRY with extensional definitions of the suspected relations, and it evaluates these in addition to the normal attribute–value terms when constructing rules. HENRY can learn relational descriptions of concepts similar to those of FOIL. Differences in output arise from the use of different heuristics for selecting "good" terms to add to rules, and the different format of the rules themselves.

The use of ripple-down rules allows HENRY to generate concept descriptions that are concise, contain little of the duplication necessary in DNF rules, and are updateable to reflect changes in the target concept. However, all the logical and deductive capabilities of DNF rules are retained, and the rules can be re-expressed in that form if necessary.

1.4 Thesis outline

The remainder of this report is in five chapters. Chapters 2 and 3 present background information on the two major components of this project. Chapter 2 gives a detailed description of ripple-down rules, discusses their benefits over other forms of concept description, and compares them with other exception-based approaches. The first section illustrates the structure of ripple-down rules, and shows how a concept description can be

built from them by either a human expert or a learning algorithm. The next section describes INDUCT and shows how it generates concept descriptions in both DNF and ripple-down rule form. The final section discusses an exemplar-based learning technique that uses exceptions, and compares its concept descriptions with ripple-down rules. Chapter 3 examines relational learning, its advantages over propositional learning, and explains how relational knowledge is represented. The first section introduces the types of relation applicable to inductive learning, and describes how they are represented in the training data and the concept description. The next section introduces inductive logic programming, giving a taste of the theoretical background, and demonstrating the types of problem applicable to this style of learning. The final section describes the relational learner FOIL, a typical inductive logic programming system, and presents some of the issues it encounters.

Chapters 4 and 5 cover the practical aspects of the project. Chapter 4 presents the implementation of HENRY. The first section deals with specifics of the implementation and extension of the INDUCT algorithm, and describes HENRY's use of propositional and relational data. The second section illustrates the various ripple-down rule output formats and describes the uses of each one. The final section gives an overview of ABE, HENRY's partner in evaluating the ripple-down rules on test data. Chapter 5 presents and discusses an experimental evaluation of the INDUCT algorithm, and compares relational learning algorithms on several classification tasks.

Chapter 6 discusses the project's conclusions and areas for future work. Particular attention is paid to the suitability of ripple-down rules to relational learning, characteristics of relational learning tasks presented in the literature, and shortcomings of the current implementation of HENRY.

2.

Concept descriptions express the information required to differentiate classes of a particular concept. They may be used to summarise a set of instances, describe a method for generating or identifying instances, or provide an explanation for the classification of an instance. Concept descriptions can take many forms—in machine learning common types include rules and decision trees. These are explicit descriptions of concepts, and are matched against examples one attribute at a time. Some algorithms use the instances themselves as the concept description, and compare new examples using distance metrics that utilise many attributes at once.

Knowledge-based systems are a major application for concept descriptions. Figure 2.1 shows a description for the concept *iris* in the form of a set of rules. An expert system might use this description to predict which iris cultivar is represented by a new example. Typical knowledge-based systems employ large structures with many rules. For example, Langley and Simon (1995), in their presentation of fielded machine learning applications, describe a system used by British Petroleum to help configure oil and gas separation vessels. The system was developed from 1600 examples, and contains 2500 rules organised into twenty-five sets. Maintaining and updating such systems is an ongoing requirement, but the complex interactions between rules mean that small changes at one point in the concept description may have major effects elsewhere (Compton and Jansen, 1990). Even a relatively small rule set, such as that in Figure 2.1, can be difficult to update. Suppose a new iris flower was found, with dimensions *petal length* = 2.6 cm, *petal width* = 0.2 cm, *sepal length* = 5.1 cm, *sepal width* = 3.5 cm, and an expert declared it an example of the cultivar *iris setosa*. If this flower was classified by the rule set in Figure 2.1 it would be misclassified by three of the rules. Concept descriptions generated from expert knowledge, or automatically from a limited supply of data, often conflict with new rules added to account for new examples. Tools are available that discover such inconsistencies, but re-engineering the knowledge base to overcome these problems, while retaining the meaning intended by the expert, is difficult. In the iris example three of the rules require modification to allow the new example to be classified correctly. Changing the bounds for the attribute–value tests in these rules may not suffice because the examples used to create the rule set may then be misclassified by it.

```
  "Iris-setosa" IF "petallength" < 2.45
  "Iris-versicolor" IF "sepalwidth" < 2.10
  "Iris-versicolor" IF "sepalwidth" < 2.45 AND "petallength" < 4.55
  "Iris-versicolor" IF "sepalwidth" < 2.95 AND "petalwidth" < 1.35
  "Iris-versicolor" IF "petallength" >= 2.45 AND "petallength" < 4.45
  "Iris-versicolor" IF "sepallength" >= 5.85 AND "petallength" < 4.75
  "Iris-versicolor" IF "sepalwidth" < 2.55 AND "petallength" < 4.95 AND
"petalwidth" < 1.55
```

```
  "Iris-versicolor" IF "petallength" >= 2.45 AND "petallength" < 4.55 AND
"petalwidth" < 1.55
  "Iris-versicolor" IF "sepallength" >= 6.55 AND "petallength" < 5.05
  "Iris-versicolor" IF "sepalwidth" < 2.75 AND "petalwidth" < 1.65 AND
"sepallength" < 6.05
  "Iris-versicolor" IF "sepallength" >= 5.85 AND "sepallength" < 5.95 AND
"petallength" < 4.85
  "Iris-virginica" IF "petallength" >= 5.15
  "Iris-virginica" IF "petalwidth" >= 1.85
  "Iris-virginica" IF "petalwidth" >= 1.75 AND "sepalwidth" < 3.05
  "Iris-virginica" IF "petallength" >= 4.95 AND "petalwidth" < 1.55
```

Figure 2.1 A disjunctive normal form rule set describing the iris concept that might be used in an expert system.

This example illustrates an important point: additions to knowledge-based systems are usually made in the context of a previous erroneous inference. An expert is consulted to provide an explanation for the exception to the standing rule, and this explanation must be added to the system avoiding the problems outlined above. Compton and Jansen (1990) describe a knowledge structure that allows changes to be made to the knowledge base in the context of the faulty rule. By limiting the scope of the change to the area where the problem exists, *ripple-down rules* significantly reduce the time and effort required to make the alteration, and ensure its consistency. Gaines and Compton (1995) cite an example system where the use of ripple-down rules increased the rate of development of the rule base from two rules per day to ten rules per hour.

2.1 Ripple-down rule structure

The reasoning behind ripple-down rules (RDRs) is to provide a facility for expressing *exceptions* to existing rules, rather than re-engineering the entire rule set. Instead of changing the bounds of the attribute–value tests in the three rules that misclassify the new iris flower, an exception handling rule could be added to each one. The expert could be consulted to explain why the new flower violates the three existing rules, and the explanations used to extend these rules only. For example, the fifth rule misclassifies the new iris setosa as an example of the cultivar iris versicolor. Instead of altering the bounds on any of the inequalities in the rule, an exception can be made based on some other attribute:

```
"Iris versicolor" IF "petallength" >= 2.45 AND "petallength" < 4.45

        EXCEPT "Iris setosa" IF "petalwidth" < 1.00
```

This rule says that the cultivar of a new iris flower is iris versicolor if its petal length is at least 2.45 cm and less than 4.45 cm *except* if its petal width is less than 1.00 cm, in which case it is iris setosa.

Utgoff (1989) applied a similar, though more complicated, technique to the ID3 decision tree learner (Quinlan, 1986), allowing it to update the existing decision tree as

new training examples were presented. Utgoff's algorithm, ID5R, restructures parts of the decision tree as new examples filter down and are misclassified. ID5R keeps track of unused attributes and values of the examples that generated a leaf node, and uses these attributes to split the leaf node when a new example with a different class arrives. The subtree is also reordered, if necessary, to keep the attribute with the greatest information gain at the root. This is a complicated operation, and may have to be performed recursively back up to the root of the tree. However, the algorithm guarantees to produce the same tree as ID3, given the same examples in any order. The ripple-down rule approach is much simpler because exceptions are only activated in the context of their parent rule. The addition of exceptions to existing rules does not affect those higher up in the concept description.

Concept descriptions using ripple-down rules

In addition to providing exceptions in existing rule sets, RDRs can be used to represent the entire concept description at the time it is created. Gaines and Compton (1995) describe a medical diagnostic system that was reconstructed using only RDRs—part of the structure is shown in Figure 2.2. The first stage in the generation of this structure is the choice of a *default* classification for examples that fail to activate any exceptions inside the RDR structure, and fall through to the bottom. The rule at the top of the structure is left empty to ensure that all examples will at least be given the default classification. From here an expert (or an inductive learning scheme as described in the next section) can inspect examples of the concept, and decide how rules and their exceptions should be formulated. For example, in Figure 2.2 the diagnosis 00 will hold if all examples are of that type. However, when an example of type 46 is presented, the expert must decide which features distinguish it from the examples presented previously. In this instance the expert submits Rule 1.46 as an exception to the empty rule 0.00. Now, any example matching rule 1.46 will be given diagnosis 46 and all others diagnosis 00. Next, the expert may encounter an example of type 32 that fails to match rule 1.46. Another exception to rule 0.00, rule 2.32, is created to accommodate this new class. Continuing in this fashion, the structure can contain exceptions to exceptions, to any level.

Figure 2.2 shows that the RDR structure is a form of *binary tree*. The left child of a node is activated if the rule at the node is true for an example; the right child is activated if the rule is false. If an example fails to match a rule at a leaf node, the classification is obtained from the leaf node's most recent ancestor that matched the example (Figure 2.3). During classification of a new example the system can note the classification given by each rule that matches the example, and emit the most recent such classification when a leaf node is encountered.

The structure of ripple-down rules permits the expert to be as general or as specific as they desire at any stage of the construction process. This allows the concept description to mimic the way the expert sees the particular classification problem. If the expert chooses not to differentiate all examples in a multiple-class situation, an additional *if-true* rule can be added to defer the decision to the next node. This might be done to produce a pair of rules that the expert considers simpler, or more intuitive, than a single longer rule, and can be important when the system must provide an explanation for a classification. A trace of the rules activated by an example shows the expert's line of thought in classifying similar examples. Standard rule sets, in contrast, can provide no such trace, as there is no significance to the order of the rules.

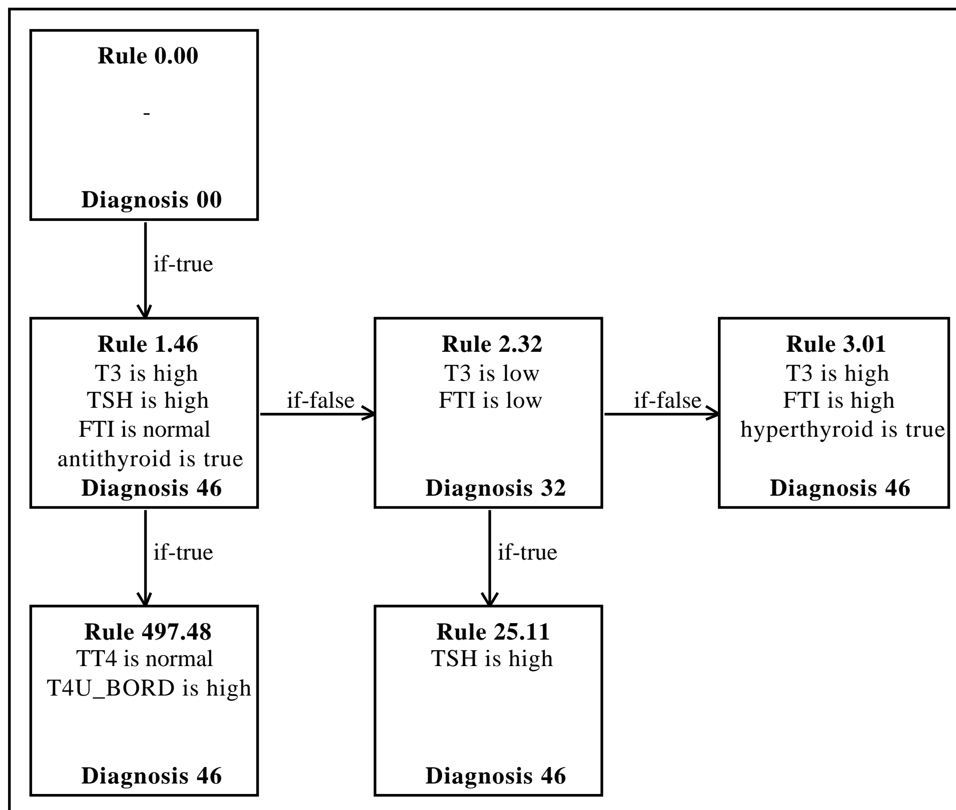


Figure 2.2 Part of a ripple-down rule structure used in a medical diagnosis system (from Gaines and Compton, 1995).

Rule structure

The rules presented so far have the form

IF condition1 *AND* condition2 *AND* ... *AND* conditionn *THEN* class is ...

This style of rule contains a *conjunction* of terms, requiring all the conditions in the first part of the rule to be true before the classification can be applied. However, a knowledge engineer is not restricted to this style of rule when creating ripple-down rules. *Disjunction* may also be used within rules, requiring only one of a number of conditions to be true to match an example. The disjunctive rule

IF condition1 *OR* condition2 *OR* ... *OR* conditionn *THEN* class is ...

is true if at least one of the conditions is true.

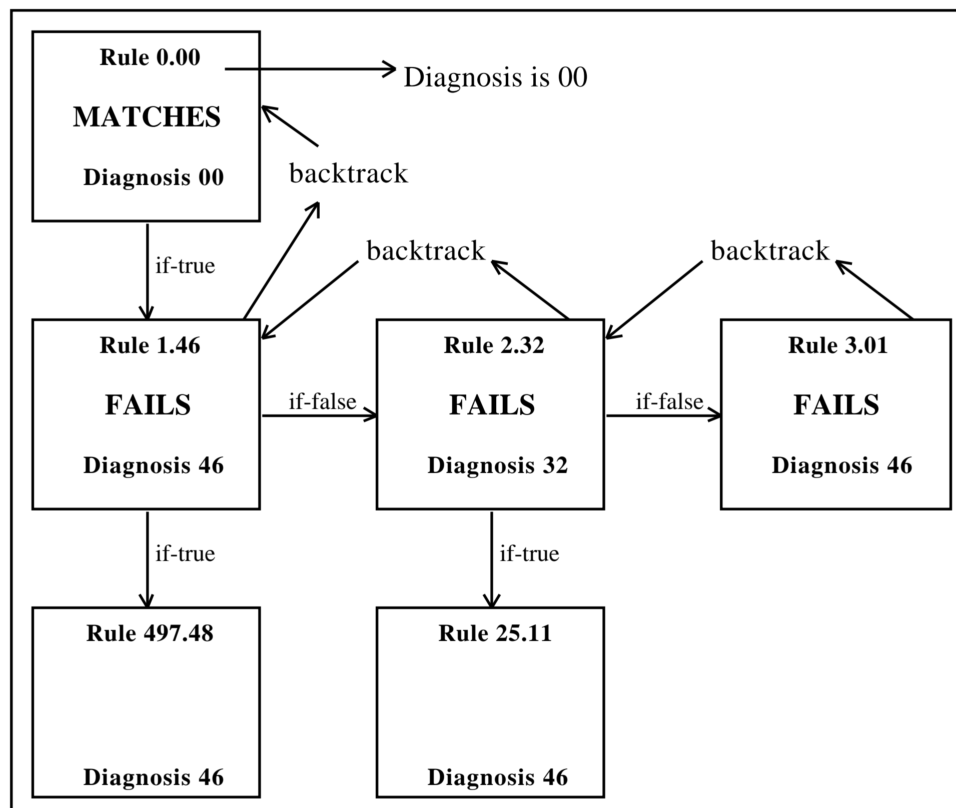


Figure 2.3 Classification of an example that fails to match rules in a ripple-down rule tree.

Mooney (1992) demonstrates that a system learning disjunctive rules generally performs at least as well as algorithms that learn decision trees and conjunctive rules. Despite this, most inductive learning algorithms generate rules with conjunctions in a style called *disjunctive normal form* (DNF)—the rule set is a *disjunction* of conjunctive rules. Mooney's algorithm learns conjunctive normal form (CNF); a *conjunction* of disjunctive rules. It is possible to generate rules that combine both conjunction and disjunction, but this increases the learning algorithm's search space substantially—although the subsetting function of C4.5 (Quinlan, 1993) performs such a role. Mooney suggests that the lack of research in developing CNF-based systems is a result of CNF having a reputation as an "unnatural" representation for knowledge.

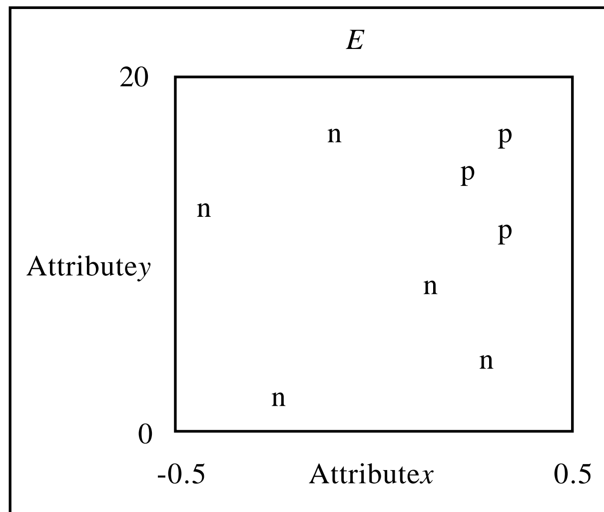


Figure 2.4 A two-dimensional instance space containing examples of two classes, *n* and *p*.

The algorithms presented in this thesis learn rules containing conjunctions of terms only. This is principally due to the fact that existing implementations of RDR learning algorithms generate rules of this form.

2.2 INDUCT

Although ripple-down rules can be updated manually, constructing the initial rule tree in this manner from a data set of many examples would be impractical. This section describes INDUCT, the first machine learning algorithm that generated ripple-down rules.

The RDR version of INDUCT is the third generation of a family of rule induction algorithms that started with PRISM (Cendrowska, 1987). Based on Quinlan's ID3 decision tree learner, PRISM uses a different style of induction to generate modular rules instead of decision trees. Whereas PRISM is a *covering* algorithm, ID3 is a *divide-and-conquer* algorithm.

Approaches to separating classes

Covering algorithms start with an n-dimensional space (where n is the number of attributes used to describe the examples) and carve off sections of the space containing counter-examples of the concept being learned. This process continues until the remaining volume contains only examples of the correct class. Thus, a general description of the concept (the entire space) is made more specific as new terms are added. Concept descriptions with a number of classes can be constructed by returning the examples in the removed sections back into the space, and repeating the exercise for each class. Figures 2.4–2.6 show the procedure in a two-dimensional space of examples, E. Examples are described by two attributes, one on each axis (Figure 2.4). Attribute x, on the horizontal axis, has legal values ranging from –0.5 to 0.5; attribute y, on the vertical axis, ranges from 0 to 20. A covering algorithm learning a concept description for examples in the space might start by inducing a rule to cover examples in the class p. For the first term in the rule describing this class, the algorithm splits the space at 0.1 on the x axis, removing from the subset examples that have values of x less than 0.1 (Figure 2.5). To remove the remaining counter-examples the algorithm next adds a term to the rule that splits the space at 12 on the y axis (Figure 2.6). Thus, the subset that contains only examples of class p can be described with the predicate $x > 0.1$ *and* $y > 12$, and the rule

IF $x > 0.1$ *AND* $y > 12$ *THEN example is a "p"*

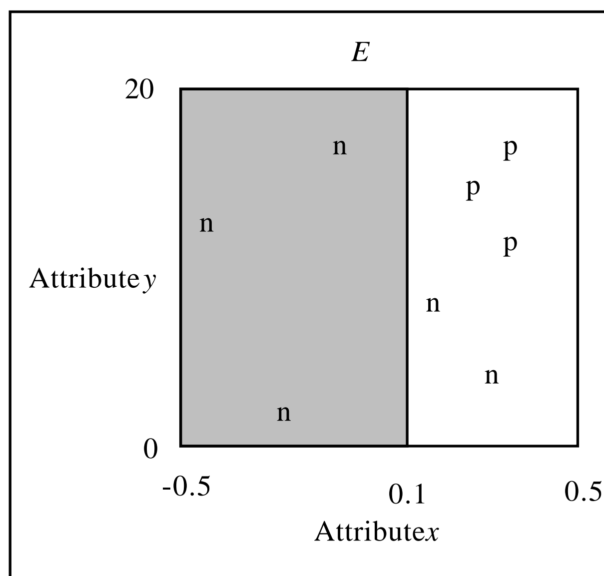


Figure 2.5 The instance space of Figure 2.4 divided by the term $x > 0.1$.

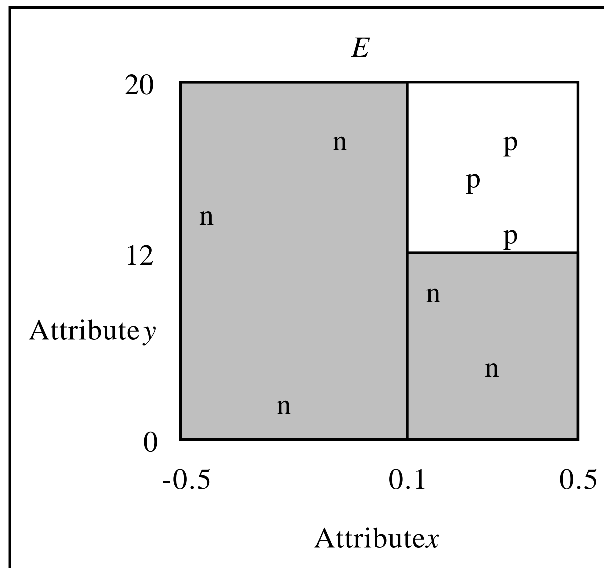


Figure 2.6 The instance space of Figure 2.5 further divided by the term $y > 12$.

can be used to decide whether new examples are also of class *p*. The process is repeated to learn a rule for examples of class *n*.

Divide-and-conquer algorithms create a concept description that covers all classes in one pass. The data is divided at each step by a test on a single attribute, and the subsets produced are further split until they contain examples of a single class. For example, a divide-and-conquer algorithm learning the above problem might first split the data set using the *x* attribute. If it used a similar performance metric to the covering algorithm it might also make the split at $x = 0.1$, and the space would be divided exactly as in Figure 2.5. However, unlike the covering algorithm's rules that specify a single class, the concept description would take both classes *n* and *p* into account. The decision tree in Figure 2.7 shows how the split creates two subsets. The subset in the left branch contains only examples of class *n*, whereas the subset in the right branch contains examples of both *n* and *p*. To split this subset and separate the different classes, the algorithm chooses the *y* attribute, selecting $y = 12$ as the split point as before. The decision tree in Figure 2.8 shows how this split expands the right subtree, which now contains a decision node. The children of this node contain examples of a single class each, *n* for the left child and *p* for the right child, and the task is complete.

The goal of both approaches is to produce a concise and accurate concept description, but they attack the problem from different directions. Divide-and-conquer algorithms try to achieve high accuracy by creating the smallest possible concept description, whereas covering algorithms try to produce a small concept description by using the most accurate terms at each step.

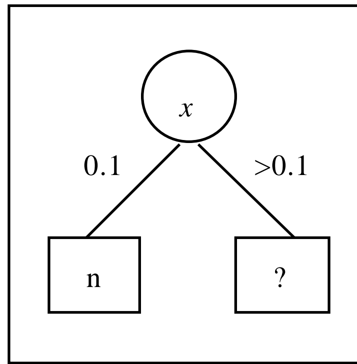


Figure 2.8 A decision tree that divides the data from Figure 2.4 into two subsets.

The two approaches, covering and divide-and-conquer, produce different forms of concept description. The "one subset at a time" policy of covering algorithms naturally produces concept descriptions as sets of rules, each rule describing a homogenous subset of examples. The recursive splitting approach of divide-and-conquer algorithms leads to a decision tree where each node represents a split in the data that reduces the "impurity" of the subset being divided.

Cendrowska considers the decision tree style of output to be one of ID3's major weaknesses. She describes decision trees as being incomprehensible to humans, difficult to manipulate by humans and computers, and difficult for providing explanations of classifications. By producing a concept description in the form of modular rules, PRISM overcomes these problems.

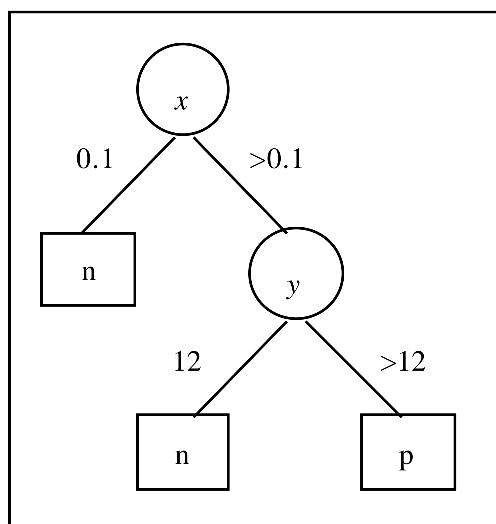


Figure 2.7 The decision tree in Figure 2.7 with the right subtree divided, this time by the attribute *y*.

```
function make_rules(Attrs, Training_set): dnf_rule_set
var    Rule: dnf_rule
       Rules: dnf_rule_set
begin
       for Class  set_of_class_values do
              while e : e  Training_set and eclass = Class begin
                     Rule.Class := Class
                     Rule.Clause := best_clause(Class, Attrs,
Training_set)
                     remove from Training_set examples of Class
                     Rules := Rules  Rule
              end while
              return examples of Class to Training_set
       done
       return Rules
end make_rules
```

A *dnf_rule_set* is a set of *dnf_rules*.
A *dnf_rule* is a structure with two components:
    Clause: A *dnf_clause.*
    Class: The class value predicted for an example that matches Clause.
A *dnf_clause* is a conjunction of *dnf_terms*.
A *dnf_term* has the form *attribute–value*.

Figure 2.9 Pseudocode outline of the PRISM covering algorithm.

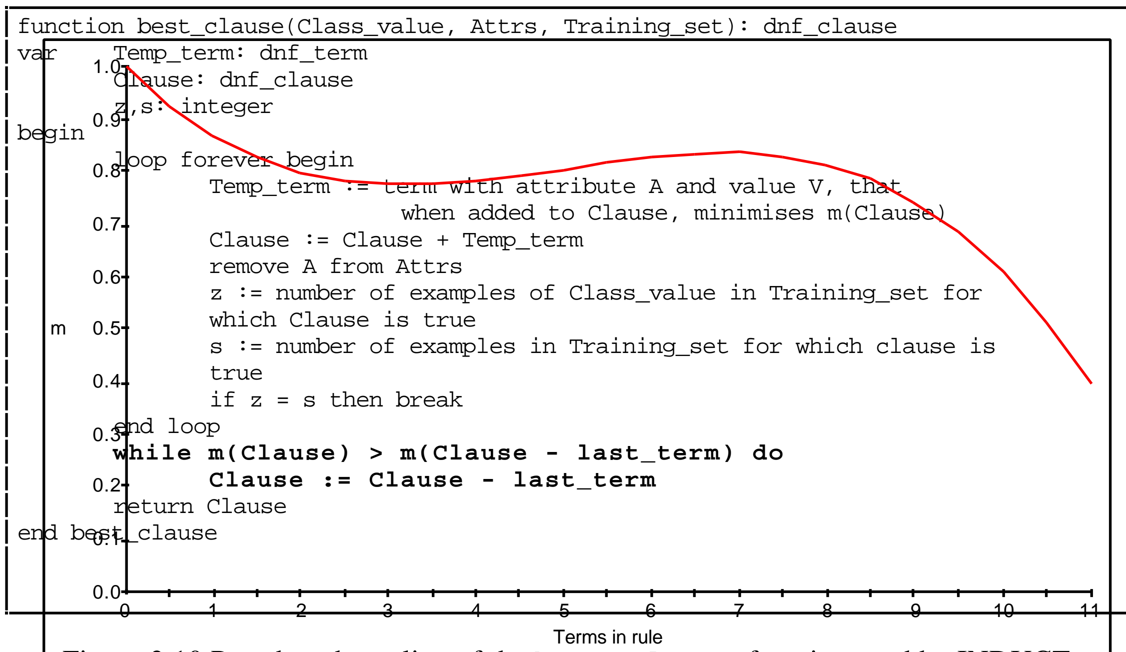
PRISM

Figure 2.9 shows a pseudocode outline of PRISM. The code consists of two nested loops—the outer loop selects a class value, and the inner loop generates rules until the class is covered. The `best_clause` function returns a conjunction of terms that covers only examples of the current class. PRISM uses a simple term selection heuristic based on the probability that an example has a particular classification given some attribute–value pair. The next term added to a rule is the one that selects the most positive examples, and the fewest negative examples. This is given by the ratio z/s, where s is the number of examples selected by the term, and z is the number of these that are positive. Terms are added until the rule selects only positive examples, that is, until $z = s$.

PRISM guarantees a rule set is *complete*—every example is covered by at least one rule—and *consistent*—every example is predicted to belong to only one class.

Statistically well-founded induction

INDUCT differs from PRISM in two respects. First, the `best_clause` function *prunes* each rule when it is completed (Figure 2.10). Working back from the end of the rule, terms are removed until the rules “quality” (measured by the m-function, described next) is maximised. Pruning allows INDUCT to improve rules that *overfit* the training data. For example, a rule may select twenty examples, all of which are positive (z/s = 20/20 = 1), and removing the last term gives z/s = 35/37 = 0.946. Although the shortened rule selects




```
function best_clause(Class_value, Attrs, Training_set): dnf_clause
var     Temp_term: dnf_term
        Clause: dnf_clause
        z,s: integer
begin
        loop forever begin
               Temp_term := term with attribute A and value V, that
                           when added to Clause, minimises m(Clause)
               Clause := Clause + Temp_term
               remove A from Attrs
               z := number of examples of Class_value in Training_set for
               which Clause is true
               s := number of examples in Training_set for which clause is
               true
               if z = s then break
        end loop
        while m(Clause) > m(Clause - last_term) do
               Clause := Clause - last_term
        return Clause
end best_clause
```

Figure 2.10 Pseudocode outline of the `best_clause` function used by INDUCT.

Figure 2.11 Curve showing how local minima in the m-function may be encountered as a rule is generated.

negative examples it is more general—a desirable feature in machine learning. Incomplete rules such as this may result in training examples being given several classifications, but this behaviour is acceptable because accuracy is expected to increase on new examples.

Pruning, instead of stopping a rule when its quality begins to decrease, allows INDUCT to overcome local maxima that may be present in the search space (Figure 2.11). If the algorithm stopped the rule at three terms because the m-value of the next term increased (a low m-value suggests high rule quality), it would miss the continuation of the downward trend at the seventh term. If it stops adding terms before reaching the seventh term because the rule is complete, it can prune back to the third term to minimise the m-value.

Figure 2.12 shows the basis for INDUCT's statistical test: the m-function. The set E is the universe of examples, the subset Q is defined by the predicate to be learned (the target predicate), and the subset S is one of many defined by the test predicates being evaluated. The intersection of Q and S contains examples of the target predicate that are selected by the test predicate, and will be closest to Q for the test predicate best describing the target predicate. The heuristic used to evaluate competing test predicates compares each with a random selection of the same number of examples, and asks "what is the probability that a random selection of the same number of examples would achieve the same or greater accuracy?" An algorithm using this heuristic attempts to identify the test predicate that is the least likely to have arisen by chance.

The formula used to calculate the probability that if $s = |S|$ examples are selected at random, without replacement, exactly $z = |Q \; S|$ of them are in class c (defined by Q), is

$$P = \frac{{}^{k}C_{z}\,{}^{n-k}C_{s-z}}{{}^{n}C_{s}},$$

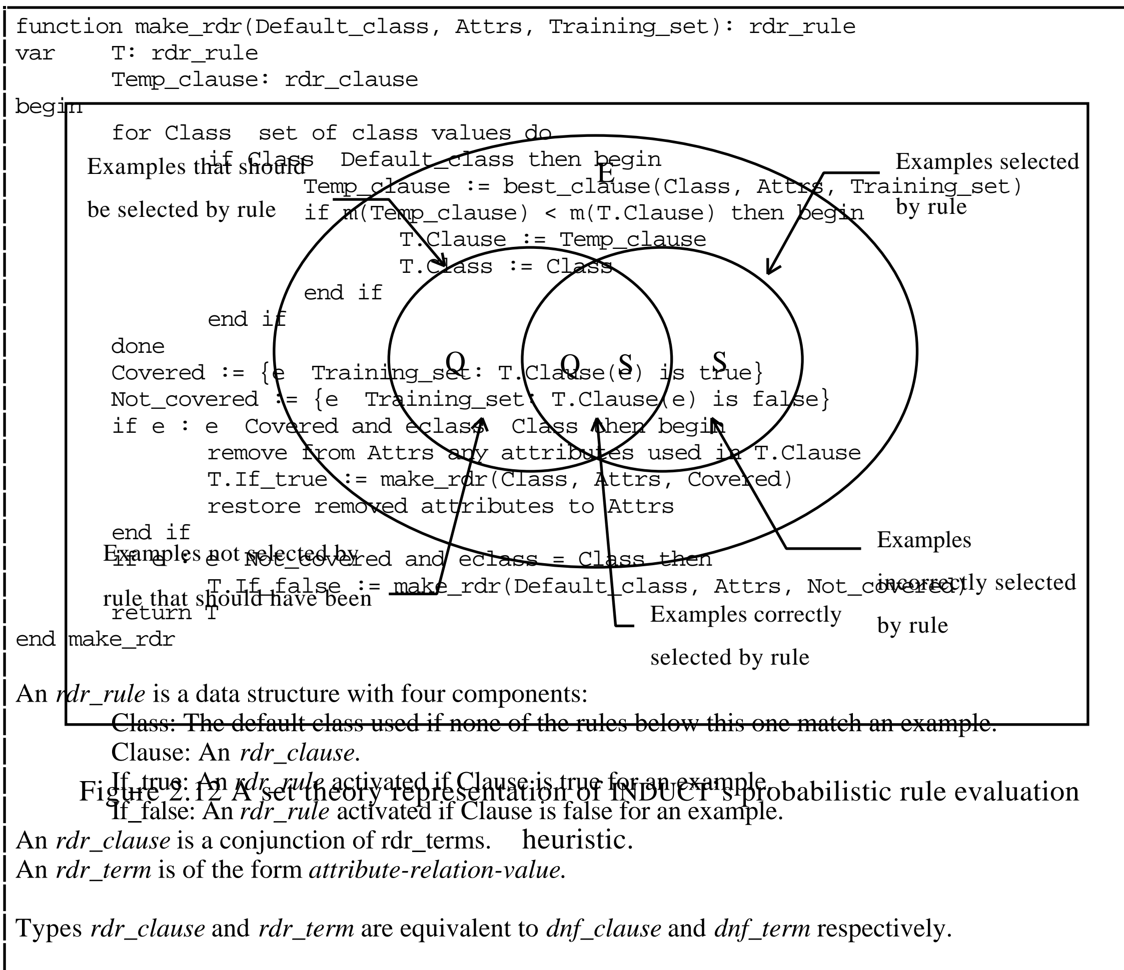


Figure 2.12 A set theory representation of INDUCT's probabilistic rule evaluation heuristic.

```
function make_rdr(Default_class, Attrs, Training_set): rdr_rule
var     T: rdr_rule
        Temp_clause: rdr_clause
begin
        for Class  set of class values do
                if Class  Default_class then begin
                        Temp_clause := best_clause(Class, Attrs, Training_set)
                        if m(Temp_clause) < m(T.Clause) then begin
                                T.Clause := Temp_clause
                                T.Class := Class
                        end if
                end if
        done
        Covered := {e  Training_set: T.Clause(e) is true}
        Not_covered := {e  Training_set: T.Clause(e) is false}
        if e : e  Covered and eclass  Class then begin
                remove from Attrs any attributes used in T.Clause
                T.If_true := make_rdr(Class, Attrs, Covered)
                restore removed attributes to Attrs
        end if
        if e : e  Not_covered and eclass = Class then
                T.If_false := make_rdr(Default_class, Attrs, Not_covered)
        return T
end make_rdr
```

An *rdr_rule* is a data structure with four components:
- Class: The default class used if none of the rules below this one match an example.
- Clause: An *rdr_clause*.
- If_true: An *rdr_rule* activated if Clause is true for an example.
- If_false: An *rdr_rule* activated if Clause is false for an example.

An *rdr_clause* is a conjunction of rdr_terms.
An *rdr_term* is of the form *attribute-relation-value*.

Types *rdr_clause* and *rdr_term* are equivalent to *dnf_clause* and *dnf_term* respectively.

Figure 2.13 Pseudocode outline of the INDUCT(RDR) algorithm.

where $k = |Q|$ and $n = |E|$, because z examples will be among the set of k examples that have class c, and $s - z$ examples will be among the $n - k$ examples that do not have class c. Of the s cases selected by S, z are in class c, so the probability that a randomly selected subset will be more accurate than S must sum for all values from z up to s (or k if S selects more examples than are in class c):

$$m(S) = \sum_{i=z}^{\min(s,k)} \frac{{}^{k}C_{i}\,{}^{n-k}C_{s-i}}{{}^{n}C_{s}}$$

However, this is expensive to calculate when there are many examples, and an approximation can be obtained using sampling *with* replacement. This gives a constant probability k/n that a selected example will be in class c. The formula

$$P = {}^{s}C_{z}\left(\frac{k}{n}\right)^{z}\left(1-\frac{k}{n}\right)^{s-z}$$

gives an approximation for m,

$$m(S) = \sum_{i=z}^{\min(s,k)} {}^{s}C_{i} \left(\frac{k}{n}\right)^{i} \left(1 - \frac{k}{n}\right)^{s-i}$$

It is this approximation that Gaines uses in his discussion of the INDUCT algorithm. In fact Gaines uses the sum

$$m(S) = \sum_{i=z}^{s} {}^{s}C_{i} \left(\frac{k}{n}\right)^{i} \left(1 - \frac{k}{n}\right)^{s-i}$$

which is incorrect when the number of examples selected by the rule exceeds the number in the class. As there is no possible random (or any other) selection that can exceed k examples of class c, the part of the sum from $s - k$ up to s is not valid. Chapter 5 presents an experimental comparison of the correct formula for m (a hypergeometric distribution) and the approximation m (a binomial distribution).

Induction of ripple-down rules

Gaines and Compton (1995) applied the statistical methodology of the INDUCT1 algorithm to the task of learning ripple-down rules. The recursive nature of the new algorithm (Figure 2.13) reflects that of the RDR structure itself. First, a default class value is selected for the top level empty rule—Gaines uses the class that occurs most frequently in the training data for this value. The algorithm then finds the rule pertaining to any class other than the default class that has the smallest m-value using the `best_clause` function of the standard INDUCT algorithm. The training set is then split into two subsets: the first containing all examples for which the clause is true, and the second containing all examples for which it is false. If either of these subsets contains more than one class—a situation made possible by pruning—the algorithm calls itself recursively on that subset. The call on the set of examples covered by the rule uses the class value of the rule. For the other subset, the existing default class is used again.

2.3 Nested Generalised Exemplars

Standard rule sets without exceptions divide the instance space of a concept with axis-parallel planes. Each term in a rule adds another plane to the space, reducing the volume predicted to contain instances of a given class. Figure 2.14 shows two examples of two-dimensional spaces as they might be partitioned by DNF rule sets. The first diagram shows the simplest means of dividing the space—all examples with value 0.5 for attribute

---

1 Henceforth the RDR-learning version of the original INDUCT algorithm shall be refered to as INDUCT(RDR) and the DNF version as INDUCT(DNF).

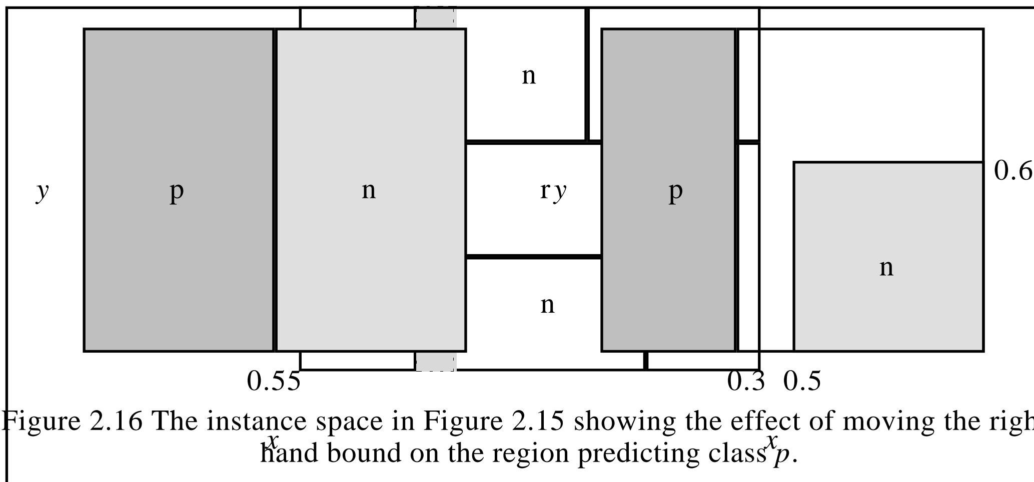


Figure 2.16 The instance space in Figure 2.15 showing the effect of moving the right-hand bound on the region predicting class *p*.

Figure 2.14 Two example instance spaces divided by DNF rule sets.

x are predicted to be in class *p*, and all those with a value > 0.5 are predicted to be in class *n*. This division can be made with a very simple rule set:

IF x 0.5 THEN p

IF x > 0.5 THEN n

The second diagram shows a more complex situation. The rule set defining these regions is:

IF x 0.3 THEN p

IF x > 0.5 AND y 0.6 THEN n

These two rules correctly predict examples falling within the two shaded areas only. Examples falling in the unshaded region are not classified by either of the rules. In this situation the bounds of a region with the same class as a new example can be moved so the example falls inside the region. INDUCT learning ripple-down rules does not allow this “empty” space to occur—all examples falling outside the scope of the rules are given the default prediction.

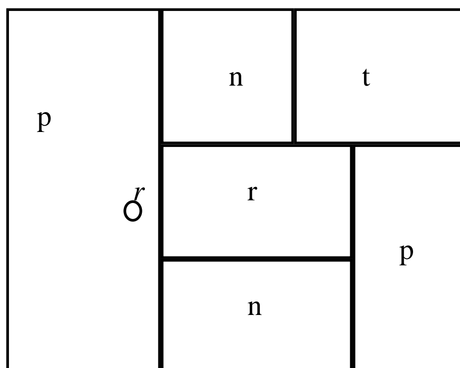


Figure 2.15 An instance space divided into regions predicting four classes.

Figure 2.15 shows how a space may be completely divided up to predict four different classes. When a new example is misclassified by one of these regions the bounds of that region must also be altered to correct the prediction. If a new example of class *r* fell into the region of class *p* on the left side of Figure 2.15, the right-hand bound of that region could be moved leftward extending the central *r* region to cover the new example (Figure 2.16). However, this reduction of the region for *p* also enlarges the two regions predicting *n*, and any examples of *p* that may have fallen into these areas will now be misclassified. The RDR solution to this problem does not alter the bounds of *p*. Instead, it introduces a region inside *p* that predicts *r* (Figure 2.17), and is defined by an exception to the rule that defines *p*.

Exceptions in exemplar-based learning

Introducing exceptions into existing regions is not unique to ripple-down rules. Salzberg (1990) presents Nested Generalised Exemplars (NGEs), a theory for learning where the predicting regions in the instance space are specified by actual examples rather than as rules. Salzberg's NGE learner, called EACH (Exemplar-Aided Constructor of Hyper-rectangles), falls into the field of exemplar-based learning, where new examples are classified by the example or examples that are most similar according to a measure of the distance between examples in the instance space. The most popular metric measures Euclidean distance in n-space, with all attributes normalised to eliminate the bias of different scales (Aha *et al*., 1991). Figure 2.18 shows how a classification is made using this *nearest neighbour* technique in two-dimensional space. The new example is closest to an example of class *p,* so it is given this classification. Cleary and Trigg (1995) discuss the use of *entropy* as an alternative measure of distance.

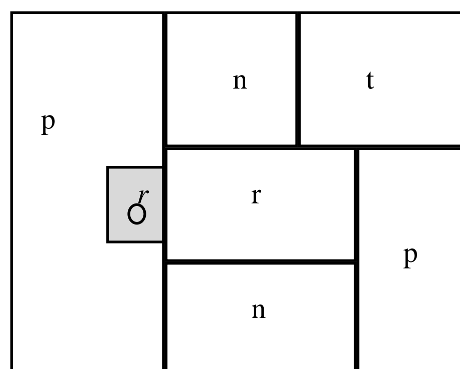


Figure 2.17 The instance space of Figure 2.15 with an exception region surrounding the new example of class *r*.

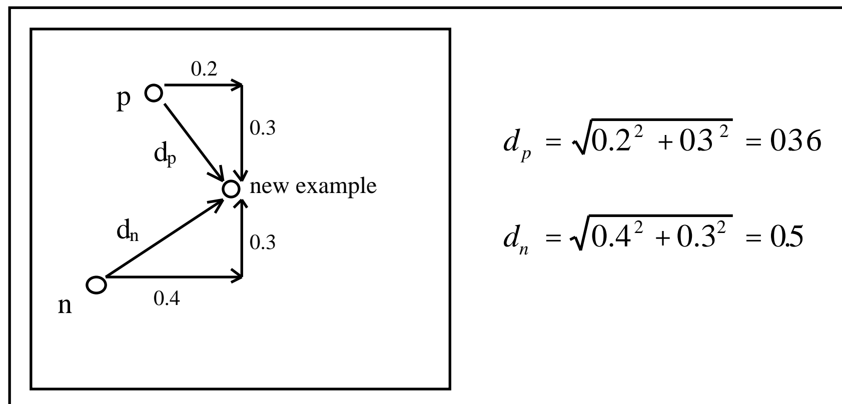


Figure 2.18 A Euclidean distance function used to determine the stored example most similar to the new example.

NGE differs from the *instance-based* approach of Aha's IB algorithms, and Cleary and Trigg's K*, which store examples as points in the instance space. Instead, it allows EACH to *generalise* examples to form predictive regions similar to those defined by rules. "Filling in" the space between examples produces axis-parallel *hyper-rectangles* (Figure 2.19), which along with point instances that have not been generalised, are called *exemplars,* because they represent characteristics of previous examples without necessarily storing them explicitly. In instance-based learning exemplars are actual examples.

New examples falling inside hyper-rectangles are predicted to have the class of the examples that bound the region. Figure 2.20 shows an example is classified as *p* despite being nearer the stored example of class *n* than either of the points defining the encompassing hyper-rectangle. Examples falling in the space between exemplars are classified using a Euclidean measure of the distance to the nearest plane of any hyper-rectangles and point of any single-instance exemplars. The region of influence generated by this technique is illustrated in Figure 2.21. The diagram shows a new example classified

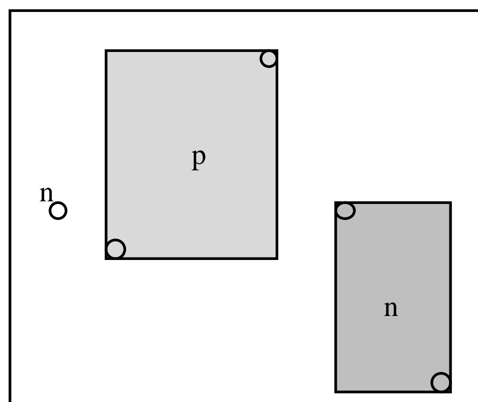


Figure 2.19 An example two-dimensional instance space with two hyper-rectangular exemplars and one point exemplar.

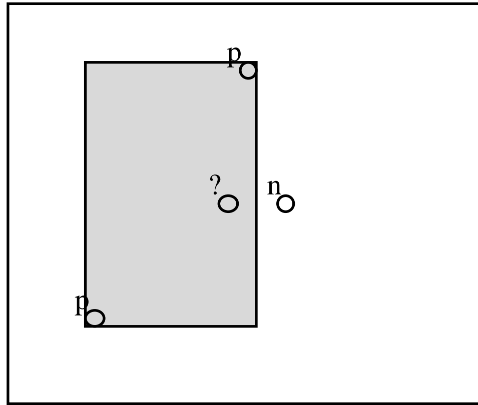


Figure 2.20 The new example is classified as a *p* because it falls within the hyper-rectangle.

as an *n* by an instance-based scheme, measuring the distance from each stored example, is classified as a *p* by an NGE-based algorithm because it is nearer to the edge of the hyper-rectangle than the point exemplar.

If EACH misclassifies a new example that falls outside the existing hyper-rectangles, the exemplar with the correct class value that lies closest to the example is extended to cover it. If that exemplar is a single point a new hyper-rectangle is created from the two examples.

If a new example that falls inside a hyper-rectangle is correctly classified, the exemplar is left unchanged, and the new example is discarded because it provides no new information about the concept. If the new example is misclassified, it is stored as a point inside the exemplar (Figure 2.22). This single example cannot be used for classification on its own because any new examples falling near it in space will be swallowed up by the encompassing hyper-rectangle. Unless the new example and the exemplar have identical values for every attribute there will always be a piece of the hyper-rectangle between them, so the distance to the hyper-rectangle will always be 0. However, if there is already

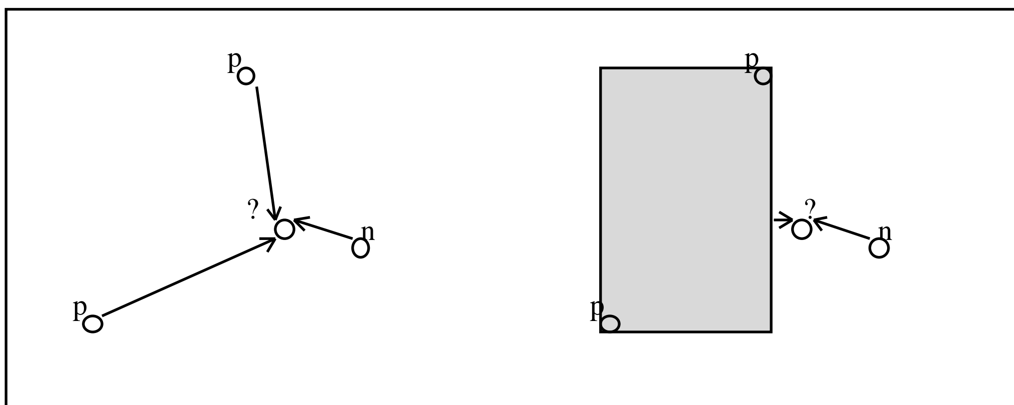


Figure 2.21 Instance-based and generalised-exemplar methods for classifying a new example.

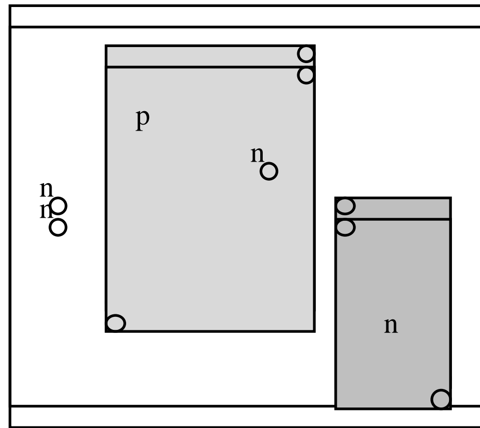


Figure 2.23 Two point exemplars in the class *p* hyper-rectangle are joined forming an exception exemplar.

Figure 2.22 Examples falling near the point exemplar are classified as class *p* because they fall inside the hyper-rectangle.

a point exemplar of the same class inside the encompassing hyper-rectangle, EACH creates an exception hyper-rectangle between the two points (Figure 2.23). The algorithm is allowed to *nest* hyper-rectangles in this manner to any degree, providing exceptions to exceptions just as in a ripple-down rule tree.

Comparing nested generalised exemplars and ripple-down rules

There are important differences between INDUCT learning ripple-down rules and EACH learning nested generalised exemplars. The first relates to the “empty” space between exemplars in NGE. This is the only place where EACH must use its distance metric, because examples falling inside an exemplar are 0 distance from it by definition. In a many-dimensional space, partitioning the space between hyper-rectangles becomes very complex. INDUCT avoids this problem by “filling” the empty space with the default class value, an action akin to EACH filling it with an exemplar before learning commences. Every new example that differed from the default class would then create exception exemplars, and no distance metric would be required. However, choice of a default class is difficult for an incremental learning algorithm such as EACH. Although INDUCT can obtain class distribution information from a data set prior to learning, EACH must assume an equal distribution of classes in the training data.

The second issue involves determining the bounds for the predictive regions in the instance space, and arises from the way each algorithm imbibes training examples. EACH is an incremental learner, meaning it examines one example at a time, and may not necessarily know the ranges of values for numeric attributes, or the distribution of symbolic values such as “class”. It uses examples as the bounding points of hyper-rectangles, assuming that any new example falling between the examples defining the hyper-rectangle will have their class value. Nesting exception exemplars allows EACH to amend the concept description if this assumption is incorrect. The algorithm can extend

the bounds of existing exemplars to cover new examples, but the values of observed examples always define where these edges may be. INDUCT is not incremental, and requires all the training examples at once. This gives it the luxury of viewing all of the training examples before determining the best place to divide an attribute, and it can set a boundary in place with the assurance that all the necessary training examples will fall within that boundary.

Experimental comparison of EACH and INDUCT is presently limited to two data sets—the iris data set introduced earlier, and a data set used to predict recurrence of breast cancer. The third data set used by Salzberg was not available for testing. Although it is not possible to make a direct comparison between the size and structure of concept descriptions, the accuracy attained by the two algorithms is similar. On the iris data set EACH achieved between 88% and 95% accuracy (depending on the setting of a feature adjustment rate) using a random selection of 66% of the data for training, with the remainder used for testing. INDUCT averaged 94% accuracy using the same sample sizes over twenty-five runs. On the breast cancer data EACH achieved between 69% and 73%, using 70% of the data for training, and INDUCT achieved 70%, again over twenty-five runs.

These results indicate that there is little difference in performance between NGE and RDR. However, further investigation is necessary to determine the effects of NGE's distance metric and RDR's default class approach to the "empty space" issue.

An important point, in the context of this project, is that ripple-down rules can be extended to include relational terms. This opens up the RDR technique to an array of classification problems for which NGE's attribute–value-only approach is inappropriate.

3.

Relational Rules

The rules generated by INDUCT and PRISM contain terms of the form *attribute–relationship–value*. These algorithms use the "equal" relationship because they learn from symbolic data, but any binary relationship can be applied. An example matches a term if its value for the term's attribute has the specified relationship with the term's value. For example, the numeric term *height > 160.05 mm* would be true for a pygmy marmoset

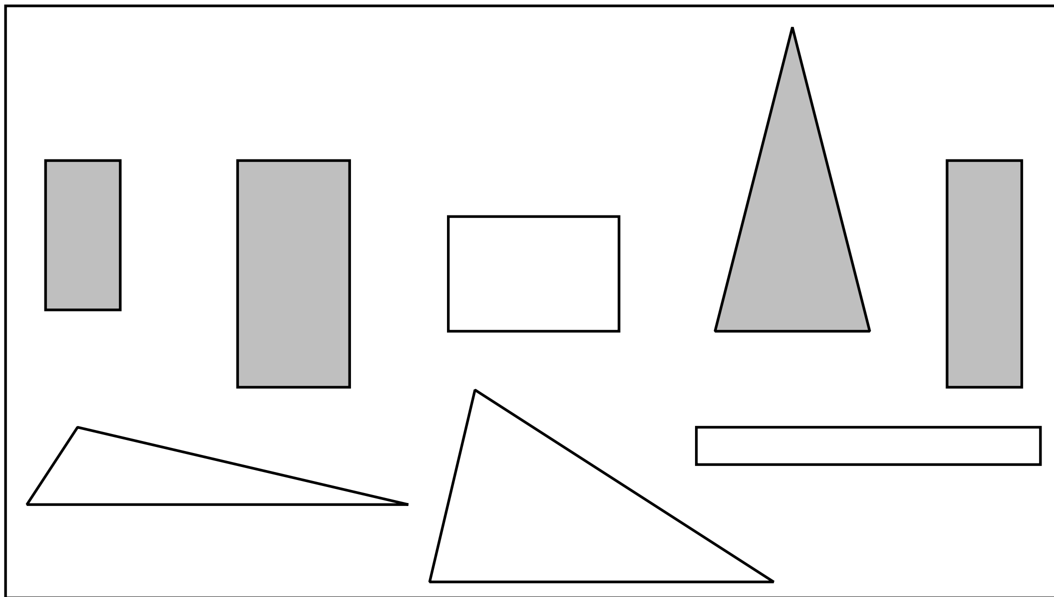


Figure 3.1 Eight blocks representing the "standing up" concept.

that was 174 mm tall, and false for a stock cube that was 20 mm high. With symbolic attributes equality can be used as the relationship. The term *height = tall* might be true for a giraffe, and false for a guinea pig. Often the meaning of symbolic values, such as "tall", is only valid in the context of the concept, or attribute, in question. In a data set consisting solely of guinea pigs, it might be possible to refer to some as "tall".

Rules that make use of such terms are *propositional* because the attribute–value language used to define them has the same expressive power as *propositional calculus* (Lavrac and Dzeroski, 1994). In many classification tasks propositional rules are sufficiently expressive for concise, accurate concept descriptions. The iris concept, for example, is well described by propositional rules. However, there are situations where a more expressive form of rule would provide a more intuitive and concise concept description. Suppose we have a set of eight building blocks of various shapes and sizes (Figure 3.1), and we wish to learn the concept "standing up". This is a two-class problem with class values "standing" and "lying." The four shaded blocks are positive (standing)

```
@relation standing_up
@attribute width integer
@attribute height integer
@attribute edges integer
@attribute class {lying,standing}
@data
2,4,4,standing
3,6,4,standing
4,3,4,lying
7,8,3,standing
7,6,3,lying
2,9,4,standing
9,1,4,lying
10,2,3,lying
```

Figure 3.2 An ARFF file containing information about the width, height, and number of edges of the building blocks in Figure 3.1.

```
"lying" IF width >= 3.50 AND height < 7.00
"standing" IF height >= 3.50
```

Figure 3.3 Two propositional rules describing the "standing up" concept generated from the data file in Figure 3.2.

```
1,2,4,'standing'
```

Figure 3.4 A block misclassified by the rules in Figure 3.3.

examples of the concept, and the unshaded blocks are negative (lying) examples. The only information the learning algorithm will be given is the width, height, and number of sides of each block. Figure 3.2 shows this data in ARFF format (Garner *et al*., 1995). The first line of the file gives the name of the target concept—in this case "standing_up". This is followed by four lines defining the attributes used to describe examples of the concept. The first three are numeric, giving dimensional information about each block; and the fourth is the value to be predicted using some function of the first three.

Running INDUCT(DNF) on this data set produces the rules in Figure 3.3. This rule set is concise and reasonably accurate, remembering that INDUCT will have pruned the rules to prevent overfitting, classifying only one block (the large triangular block at bottom centre) as both "standing" and "lying". However, the rules are intuitively bad—the block in Figure 3.4 would not be classified by either rule, and it is easy to devise many legitimate blocks that would "fit" the data set but not the rules. One solution to this problem is to introduce new attributes that contain more information about the blocks, but it is difficult to see how to do this without just repeating the class value.

Another solution is to let the learning algorithm examine more than just the values of the attributes independently. A human classifying the eight blocks would probably say "the standing blocks are those that are taller than they are wide". This "rule" does not compare attributes with specific values—it compares the attributes with each other:

> "If the value for the attribute *height* is larger than the value for the attribute *width* then the block is standing",

or

> "If *height* is greater than *width* then the block is standing".

For an example matching this rule the actual values of *height* and *width* are no longer important, as long the relationship between the two attributes is true. Rules of this form

are *relational,* because they express relationships between attributes. Section 3.2 presents a more formal definition of relations, and describes how they may be used in learning.

The new block that failed to match the propositional rules will be classified correctly by the relational rules in Figure 3.5. These rules were generated by the algorithm that produced the propositional rules, suitably extended to search for relationships such as "attribute x is always larger than attribute y" in the training data.

```
"lying" IF width is-bigger-than height
"standing" IF width is-smaller-than height
```

Figure 3.5 Two rules that describe "standing up" concept that specify relationships between the attributes of the blocks.

3.1 Learning relations

To use relations for comparing attributes a learning algorithm can exploit knowledge implicit in the data type of each attribute. In the "standing up" example the learning algorithm was able to use the "is bigger than" relation on *width* and *height* because they are numeric attributes. It would not make sense to use the same relation on symbolic attributes such as colour or texture. Comparing a symbolic attribute and a numeric attribute is also invalid because there is no sensible relationship that can apply to both numbers and symbols.

Standard relations

If no other relations are supplied the following may be used on attributes of the same type:

< and (or and >) for numeric attributes,

and

= for symbolic attributes.

Incorporating these three relations into INDUCT(DNF) requires an addition to the `best_clause` function. It is now required to search for terms with two attributes of the same type, and a relation applicable to those attributes[2]. The alteration is small, but it enlarges the search space dramatically. The original search for a single propositional term covers a space of $a \times n$ terms in the worst case (if every example has a unique value for

---

2 Details of the alteration and its implementation are defered to section 4.1.

each attribute), where *a* is the number of attributes and *n* is the number of examples. Searching for inter-attribute relationships adds an additional $a^2 \times n$ terms.

Techniques to reduce the search space are covered in Section 3.3 on the FOIL algorithm. However, there are simple heuristics, such as the type-dependent limitations of the <, , and = operators described above, that can be applied in general. Removing the need to search through terms that compare numeric and symbolic attributes is the most fundamental space reduction method, but “type checking” of attributes can be taken further than the numeric/symbolic differentiation. Often attributes of the same type will be quite obviously incompatible for comparison. Continually checking to see whether the values of *colour* are always equal to the values for *smell* in a data set of flowers is pointless. Although both of these would be symbolic attributes, that is where the similarity ends. *Colour* would use an entirely different set of values to *smell,* so we know a priori that the term *colour* = *smell* will never arise in the rule set.

Even when the sets of values used by two symbolic attributes are identical we may not want the learning algorithm to compare them. For example, the letters *A, B, C, D*, and *E* might represent the colours *red, blue, green, yellow*, and *black* on a colour chart, and also values for the *smell* attribute. These attributes should not be compared because the assignment of the five letters is completely arbitrary. If the algorithm found the term *colour* = *smell* to be a good discriminator, it would do so purely by chance. Changing the assignment of the letters to different colours would not change the meaning of the data, but it would alter the chances of that particular term being selected again.

Incompatibilities also arise in numeric attributes. Although all numeric attributes use the same “set of values” we may not necessarily wish to compare every pair of them. In the “standing up” example the two attributes compared in the rules are measurements of distance. Here, a comparison of the two numeric attributes is legitimate because the units of measurement are the same. If the data also included a *temperature* measurement for each block we would not want the learning algorithm to compare this attribute with either *height* or *width*. The different units used to quantify these features makes a comparison invalid, and once again, the size and origin of the units is arbitrary. Changing from the Celsius scale to the Kelvin scale would not change the meaning of the data, but it could affect relationships between the values of *temperature* and the other numeric attributes.

The choice of units may also obviate relationships that genuinely exist in the data. If the unit used for *width* in the “standing up” example was the millimetre instead of the centimetre, “is bigger than” would not have been selected for the rule. The learning algorithm compares all integers on the same scale, and all those in the *width* column would be larger than those in the *height* column—there is nothing to learn from that relationship. It is possible to make more operators available to detect relationships hidden

by such things as different scales. For example, “attribute x is less than 10 times attribute y” might discover the “is bigger than” relationship in the “different units” blocks data. However, the infinite number of possible relations, and the difficulty of selecting a small set that is meaningful in any particular context, make this impractical.

User relations

In addition to the “standard” relations that can be applied to any pair of attributes of the same type, it might be appropriate to provide new relations specifically for an individual problem. Suppose we wish to classify a set of technical report abstracts as either related or not related to machine learning. Most text classification approaches would represent the abstracts as vectors of Boolean values indicating the presence or absence of each word in the collection. Figure 3.6 shows data of this form in ARFF format. Every word in the set of abstracts is an attribute, and each abstract is represented by a tuple of values indicating whether or not each word appears. A propositional learning algorithm applied to this data might produce the rules in Figure 3.7. Although these rules might appear acceptable it is easy to think of examples that would violate them. For example, a report discussing a natural language parsing algorithm would be misclassified as *pos* by the second rule. If we chose to represent the abstracts in the form illustrated in Figure 3.8, we can introduce several relations that will help produce more useful rules. Here, the positions of the words in the abstract are the attributes, and the words that can fill each position are the values. Providing the learning algorithm with the relations in Figure 3.9 allows it to search for relationships amongst the attributes and produce rules that contain relationships involving word position and order (Figure 3.10). The first rule classifies an abstract as *pos* if the word “learning” immediately follows the word “machine” anywhere in the text, and is more precise than merely requiring the presence of both words. The relation *successor* specifies a positional relationship between words that cannot be expressed with propositional rules.

```
@relation ml_document
@attribute class {neg,pos}
@attribute word#0 {C4.5, a, the, algorithm, the, learning, the, the, a, most, when,
approach, like}
@attribute word#1 {is, machine, 1995, that, ability, how, 1995, quality, common,
of, making, that, a}
@attribute word#2 {a, learning, conference, induce, to, to, conference, of,
algorithm, the, an, can, machine}
@attribute word#3 {machine, algorithm, on, decision, induce, use, on, learning,
for, trees, environmental, induce, learning}
@attribute word#4 {learning, such, machine, trees, decision, a, milling, in,
sorting, had, decision, decision, to}
@attribute word#5 {algorithm, as, learning, from, trees, milling, quality, many,
an, come, trees, making, drive}
@attribute word#6 {of, C4.5, to, a, as, machine, was, primary, array, to, and, in,
a}
@attribute word#7 {high, can, develop, set, a, can, held, schools, of, a, plants,
many, car}
@attribute word#8 {quality, induce, an, of, way, be, at, is, integers, decision,
should, areas, can}
@attribute word#9 {and, a, algorithm, data, to, very, the, very, is, that, be,
where, be}
@data
pos, C4.5, is, a, machine, learning, algorithm, of, high, quality, and
pos, a, machine, learning, algorithm, such, as, C4.5, can, induce, a
pos, the, 1995, conference, on, machine, learning, to, develop, an, algorithm
pos, algorithm, that, induce, decision, trees, from, a, set, of, data
pos, the, ability, to, induce, decision, trees, as, a, way, to
neg, learning, how, to, use, a, milling, machine, can, be, very
neg, the, 1995, conference, on, milling, quality, was, held, at, the
neg, the, quality, of, learning, in, many, primary, schools, is, very
neg, a, common, algorithm, for, sorting, an, array, of, integers, is
neg, most, of, the, trees, had, come, to, a, decision, that
neg, when, making, an, environmental, decision, trees, and, plants, should, be
neg, approach, that, can, induce, decision, making, in, many, areas, where
neg, like, a, machine, learning, to, drive, a, car, can, be
```

Figure 3.6 An ARFF file representing technical report abstracts as word-appearance tuples.

```
"class" = "pos" IF "machine" = T AND "learning" = T
"class" = "pos" IF "algorithm" = T
```

Figure 3.7 Two propositional rules that define positive examples from the set of abstracts.

Figure 3.8 An ARFF file representing the abstracts using word positions as attributes.

The terms in Figure 3.10 differ in their use of relations from those that use the three standard relations introduced earlier. The term *height is-bigger-than width* specifies the names of two attributes that are compared by the relation *is-bigger-than*. This means that no matter what value an example has for the *width* attribute, this term will be true if the value for *height* is larger than that value. The term *successor(machine, learning)* specifies two words, or attribute *values*, that are compared by the relation *successor*. Regardless of which pair of *attributes* have the values *machine* and *learning,* this term will be true if the relation *successor* is defined for them. Thus, the second example is classified as *pos* because word#1 has the value *machine* and word#2 has the value *learning*, and the relation *successor* is defined for word#1 and word#2. The sixth example has word#6 with the value *machine* and word#0 with the value *learning*, but there is no definition for *successor* with those two word positions so the example is classified as *neg*.

```
"class" = "pos" IF successor ("machine","learning")
"class" = "pos" IF successor ("induce","decision" ) AND
       successor ("decision","trees")
```

Figure 3.10 Two relational rules that define positive examples from the set of abstracts.

In general, relations can specify any number of attributes. For example, a relation describing the components of a list may have three attributes, *components (L,H,T),* where *L* refers to the entire list, *H* to the head or first element, and *T* to the remainder of the list. Constructing relations of the standard type with more than two attributes is also possible. A relation with three attributes, $a1$ $a2$ $a3$, could be used in a rule indicating that *width* is never larger than *depth,* which in turn is never larger than *height*, for some class of objects. However, as noted earlier, the infinite variety of such relations makes searching through them impractical.

Representing relations

Because relations can be specific to individual classification problems, they can be considered part of the data that represents the concept. These extra relations provide background information that propositional learners cannot use. Often the line between the background information and the "actual data" can be indistinct. Learning systems such as FOIL (Quinlan, 1990) do not distinguish the examples from the background relations. The target concept is represented by relations in the same form as the background information, and the algorithm attempts to generate a rule set that describes the concept in terms of these relations.

FOIL uses a representational form similar to PROLOG Horn clauses, and learning a

```
successor(word#0,word#1).        after(word#0,word#1).
successor(word#1,word#2).        after(word#0,word#2).
successor(word#2,word#3).        after(word#0,word#3).
...                              ...
before(word#1,word#0).           near1(word#0,word#1).
before(word#2,word#0).           near1(word#1,word#0).
before(word#3,word#0).           near1(word#1,word#2).
...                              ...
```

Figure 3.9 Extensional definitions for relations used in the text categorisation example.

```
measurement: continuous.
standing(measurement,measurement,measurement)
2,4,4
3,6,4
7,8,5
8,9,4
;
4,3,4
7,6,5
9,1,4
10,2,5
```

Figure 3.11 A FOIL input file representing the blocks in the "standing up" problem.

```
c4.5(doc1,word#0).                 to(doc5,word#2).
c4.5(doc2,word#6).                 to(doc5,word#9).
      …                            to(doc6,word#2).
                                         …
machine(doc1,word#3).
machine(doc2,word#1).              from(doc4,word#5).
      …                                  …

drive(doc13,word#5).        environmental(doc11,word#3).
      …                                  …
```

Figure 3.12 Extensional predicate definitions describing the positions of words in the technical report abstracts.

definition for a target relation resembles identifying a PROLOG-like program that produces the target relation as output. This form of learning is called *inductive logic programming*, and is discussed in the next section. Data intended for propositional learning is readily converted to a relational form that can be used by inductive logic programming systems. Figure 3.11 shows how the blocks data might be represented in a form suitable for FOIL. The class value "standing" is the target relation, and the four examples of standing blocks are the first four triples in its definition. The class value "lying" is regarded as "not standing," so the four examples of lying blocks become negative examples of *standing* and appear after the semicolon. A more complete

```
after (X,Y)  successor (X,Y).
after (X,Y)  successor (X,Z) AND after (Z,Y).
```

Figure 3.13 An intentional definition for the relation *after*.

description of the FOIL input format is given in Section 3.3.

The technical report abstract data can also be represented in relational form—Figure 3.12 shows part of such a data set. The definition *machine(doc*1,*word#*3*)* indicates that the word "machine" appears as the third word of document one. Similar relations describing the positions of every other word express the complete set of abstracts.

Relations exemplified so far have all been defined *extensionally*; that is, every combination of values for which the relation is true is defined separately. This can lead to very large sets of definitions for relatively small data sets. For example, if the technical report abstracts were all twenty-five words long there would be three hundred definitions for the relation *after*. By defining a relation in terms of other relations, the number of definitions can be greatly reduced. Such definitions are called *intentional,* and are often recursive. Figure 3.13 shows an intentional definition for *after* that reduces the number of

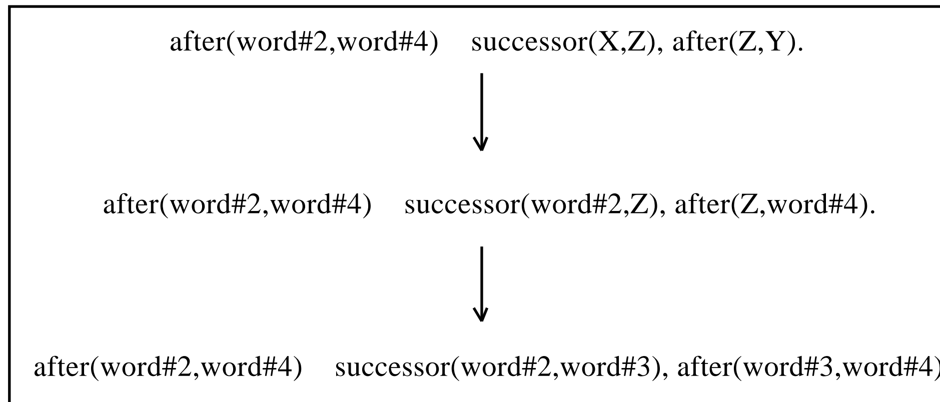


Figure 3.14 Derivation of the truth value of after(word#2,word#4) from the intentional definition in Figure 3.13.

definitions from three hundred to two. The first makes use of the extensionally defined relation *successor*, and provides a base for the second, recursive definition to build on. This simple definition shows that *successor after,* so there is no need to define both separately. The second definition uses two relations: the first selects a pair of values from the *successor* relation, and the second is a recursive call to *after* that uses a new variable (*Z*) introduced by *successor*. Figure 3.14 shows how this definition can be used to define the relation *after(word#2,word#4).* Letting *X* = *word#2* and *Y* = *word#4* the first definition fails because there is no definition for *successor(word#2,word#4)*, and it is assumed to be false under the *negation-as-failure* rule (Lavrac and Dzeroski, 1994). However, the second definition uses another variable *Z* that may be instantiated to any value. If a value can be found for *Z, after(word#2,word#4)* will be true by this definition (line 2 of Figure 3.14). The definition *successor(word#2,word#3)* from Figure 3.9 gives a value of *word#3* for *Z.* The definition is now complete (line 3 of Figure 3.14) as long as *after(word#3,word#4)* is true. There is a definition for *successor(word#3,word#4),* so this relation matches with the first definition of *after*.

3.2 Inductive logic programming

Inductive logic programming (ILP) is the intersection of inductive machine learning and logic programming (Lavrac and Dzeroski, 1994; Muggleton, 1991). Although a relatively young field it benefits from decades of research in logic and logic programming. This provides a strong theoretical basis, which accounts for much of the fields popularity, and the experimental nature of machine learning research ensures it is oriented towards practical applications.

Logic programming

variables:	X Word P4
predicate & function symbols:	b tail last4letters

Figure 3.15 Examples of valid variables, and predicate and function symbols.

Logic programming, and subsequently inductive logic programming, takes its terminology from the fields of computational logic and deductive database theory. The following definitions are taken from Lavrac and Dzeroski (1994), and build up to the syntax of the PROLOG language. They are intended to give an illustration of the depth of the theoretical background provided by computational logic, and will aid in the discussion of ILP systems such as FOIL.

Objects in a logic program are described with a *first-order alphabet* consisting of variables, predicate symbols, and function symbols. A *variable* is represented by an uppercase letter followed by zero or more lowercase letters and digits. *Predicate symbols* and *function symbols* consist of a lowercase letter followed by zero or more lowercase letters and digits. Figure 3.15 shows three examples each of variables, and predicate and function symbols.

A *term* is a variable or a function symbol followed immediately by a bracketed n-tuple of terms. A term containing a *k*-tuple has arity of *k*; a term with arity 0 is also called a *constant*. Figure 3.16 shows two terms of arity 3, and two constants.

A predicate symbol followed immediately by a bracketed n-tuple of terms is called an *atom*. A *literal* is an atom or the negation of an atom.

A *clause* has the form

$$X_1 \quad X_2 \ldots \quad X_n \left(L_1 \quad L_2 \quad \ldots L_m \right)$$

where each *Li* is a literal and *X1,...Xn* are variables occurring in *L1,...Ln*. If any of the literals are negative literals, for example $\left(L_1 \quad L_2 \quad \ldots \overline{L}_i \quad \overline{L_{i+1}} \quad \ldots \right)$, *De Morgan's Theorem* and *material implication*3 allow the clause to be written

$$L_1 \quad L_2 \ \ldots \quad L_i \quad L_{i+1} \ \ldots$$

---

3 De Morgan's Theorem, $(\sim p \quad \sim q) \quad \sim (p \quad q)$, and material implication, $(\sim p \quad q) \quad (p \quad q)$, are two of ten rules of replacement presented by Copi (1979). Using these rules derivation of the second form of the clause from the first can be performed in two steps.

terms:	append(X,Y,Z) f(g(X),h,Y)
constants:	pi() jpl7j74()

Figure 3.16 Examples of valid terms and constants.

which is commonly abbreviated to

$$L_1, L_2, \ldots \qquad L_i, L_{i+1}, \ldots$$

The quantification symbols are dropped as all variables are assumed to be universally quantified.

A *Horn clause* is a clause that contains not more than one positive literal. A *definite program clause* is a clause that contains exactly one positive literal and has the form

$$T \qquad L_1, L_2, \ldots L_n$$

where *T, L1,... Ln* are atoms. A *definite logic program* is a set of definite program clauses.

A *program clause* has the same form as a definite program clause, but with the restriction that *T* is an atom, and *L1,... Ln* have the form *L* or *not L* where *L* is an atom. A *normal program* is a set of program clauses. A *predicate definition* is a set of program clauses that have the same predicate symbol (and arity) in the positive literal. The intentional definition for *after* in Section 3.1 is a predicate definition.

Two special cases of clauses are given additional definitions. A Horn clause with no positive literal is a *definite goal*. A definite program clause with no negative literals is a *positive unit clause*. In PROLOG a positive unit clause is called a *fact*.

Database terminology

Inductive logic programming also borrows terminology from the field of relational databases. The development of *deductive databases*, which can be implemented using logic programming, allowed relations to be defined intentionally or extensionally. Therefore several definitions in database theory have counterparts in computational logic.

An *n*-ary *relation* is a subset of the Cartesian product of *n* domains, where a *domain* is a set of values. A relation is equivalent to a predicate in a logic program, but it can be *typed*, meaning the values taken by variables in a relation are limited to specific sets. This is equivalent to the search space pruning introduced in the previous section, where the learning algorithm is prevented from comparing the attributes *colour* and *smell*.

DDB terminology	LP terminology
relation name p	predicate symbol p
attribute of relation p	argument of predicate p
tuple a1,...,an	ground fact p(a1,...,an)
relation p—a set of tuples	definition of predicate p—a set of ground facts

Table 3.1 Relating database and logic programming terms (from Lavrac and Dzeroski, 1994).

A *database clause* is a typed program clause, and a *deductive database* is a set of these. Database clauses can use variables and function symbols in predicate arguments, so the language of deductive databases is more expressive than that of relational databases. The use of clauses allows relations in deductive databases to be defined intentionally, whereas in relational databases they can only be defined extensionally. The intentional definition for *after* shows that relations in deductive databases can have a very compact representation.

Table 3.1 (from Lavrac and Dzeroski, 1994) shows how the terms of deductive databases (and inductive learning) relate to those in logic programming. A data set in propositional learning is essentially a two-dimensional database, and the terminology of database theory is used in inductive learning. The ARFF file in Figure 3.2 illustrates this point. The data represents the target relation "standing_up," the columns of data are attributes, and each row is a tuple defining an instance of the relation. It is easy to see how this data could be presented as a set of extensional predicate definitions.

Empirical inductive logic programming

Inductive logic programming algorithms, such as FOIL and GOLEM (Muggleton and Feng, 1992), fall into the discipline of *empirical ILP*, and their task can be expressed as:

> Given a set of training examples *E* consisting of true (*E*+) and false (*E–)* ground facts of an unknown predicate *p* (called the *target predicate*), a description language *L* specifying syntactic restrictions on the definition of *p*, and background knowledge *B* defining predicates *qi* which do not include *p* and may be used in the definition of *p*, find a definition *H* for *p* expressed in *L* such that *H* is complete and consistent with respect to *E* and *B* (Lavrac and Dzeroski, 1994, p. 30).

The concepts of consistency and completeness are the same as those introduced during the description of PRISM, and can be defined more formally using the notion of *coverage*. Empirical ILP employs *extensional coverage* where the background knowledge

is restricted to being specified extensionally. If background knowledge is supplied intentionally it can be replaced with all true ground facts that can be derived from the it using SLD-resolution (Lloyd, 1987). The set of ground facts derivable from background knowledge *B* is called a ground model *M* of *B*.

First we define *intentional coverage* where background knowledge can be defined intentionally. Given background knowledge *B,* hypothesis *H*, and examples *E*, *H* is said to cover example *e E* if *B H* logically entails *e*,

covers (B,H,e) = true if B H e,

The function *covers (B,H,E)* is thus defined

covers (B,H,E) = {*e E* / *B H e*}.

A hypothesis *H* is complete if *covers (B,H,E+) = E+,* and consistent if *covers (B,H,E–) =* Ø.

Extensional coverage with respect to a ground model *M* restricts this definition to

coversext (M,H,e) = true if *c = T Q H* and a substitution , such that *T = e* and *Q =* {*L1,...,Ln*} *M.*

Suppose the background knowledge *B* contains the predicate definitions in Figure 3.17. This set of predicates produces the model shown in Figure 3.18. An empirical ILP system might produce a hypothesis *H* that includes the clause

c = p (X,Y,Z) r(X), s(Y), t(Z).

To see if *H* covers the example *e = p (a,b,c)* use the substitution = {*X/a, Y/b, Z/c*}. From the definition of extensional coverage *T = p (X,Y,Z),* and *T = p(a,b,c) = e*. Also *Q =* {*L1, L2, L3*} *= r(X), s(Y), t(Z),* and *Q =* {*r(a), s(b), t(c)*} *M*. Thus, *e* is covered by *H*.

Concept learning can be viewed as a search problem, and a learner can be described in terms of the structure of its search space and its search strategy (Lavrac and Dzeroski, 1994). In inductive logic programming the search space is specified by the language of logic programs, or a subset of it, which places syntactic restrictions on the dimensions of the space. The next section describes FOIL, which restricts the search by allowing only function-free Horn clauses.

```
l(a).      r(X)l(X).
l(e).      s(X)m(X).
m(b).      t(X)n(X).
m(f).
n(c).
n(g).
```

Figure 3.17 Intentional and extensional predicate definitions providing background knowledge for a relational learning task.

```
l(a).      r(a).
l(e).      r(e).
m(b).      s(b).
m(f).      s(f).
n(c).      t(c).
n(g).      t(g).
```

Figure 3.18 The set of ground facts derived from the predicate definitions in Figure 3.17.

3.3 FOIL

This section describes the empirical ILP system FOIL (Quinlan, 1990), and provides an overview of a typical relational learner. Points covered include FOIL's search strategy and heuristics, and language and data restrictions. Several example classification problems are also presented.

FOIL learns function-free Horn clauses from data expressed as relations. Given an extensional definition of a target relation expressed as a set of ground facts, the algorithm employs the covering method of induction to generate an intentional definition using the ground examples and other user-defined background relations. FOIL was designed to overcome the limited expressiveness of inductive learning algorithms that use propositional languages for describing objects and concept descriptions. It combines search techniques from propositional learning algorithms with a first-order description language that allows it to learn concepts outside the scope of propositional systems.

Covering algorithms have been described earlier so this section focuses on particular aspects of FOIL such as input and output representation, selection of literals, and restrictions on the search space. These issues are important for all empirical ILP systems—and inductive learning algorithms in general—and FOIL provides successful and often-cited responses to them (for example, Dzeroski and Lavrac, 1992; Lavrac and Dzeroski, 1994; Mooney, 1992; Pazzani and Kibler, 1992; Pazzani et al., 1992).

Defining the search space

The search space of any learning algorithm is defined by language it uses to describe objects and concept descriptions. FOIL uses the language of function-free Horn clauses to represent concepts. Relations constituting the input to the program are defined extensionally, and are a subset of this language. The search space encompasses all literals of the form

$$P(X_1, X_2, \ldots, X_m) \quad L_1, L_2, \ldots, L_n$$

where each *Li* has one of four forms

$$X_j = X_k\,,\ X_j \quad X_k\,,\ Q(V_{1,} V_2, \ldots, V_r)\,, \text{ or } \neg Q(V_{1,} V_2, \ldots, V_r)\,,$$

```
block:a1,b1,c1,a2,b2,c2,a3,b3,c3,a4,b4,c4.

arch(block,block,block)
1,b1,c1
4,b4,c4
;
2,b2,c2
3,b3,c3
.
*supports(block,block)
b1,a1
c1,a1
b3,a3
c3,a3
b4,a4
c4,a4
.
*touches(block,block)
a1,b1
b1,a1
a1,c1
c1,a1
a2,b2
...
```

Figure 3.19 Part of a FOIL input file for the “arches” problem.

where the *Xi* s are existing variable, the *Vi* s are existing or new variables, and *Q* is some relation.

Figure 3.19 shows part of a FOIL input file4 for the classic “arches” problem (see Section 5.2). The file describes twelve blocks that make up four objects, two of which are arches and two not. Figure 3.20 shows how the blocks are arranged. The input file defines seven relations, starting with the target relation *arch,* which has three attributes all of type *block*. The first line of the file defines the type *block* as twelve specific objects. The definitions given under *arch* are tuples describing the two combinations of blocks that constitute valid arches—these are positive examples of the concept. The two tuples following the semicolon describe the two combinations in Figure 3.20 that are not arches—these are negative examples. If the negative examples are omitted FOIL uses the *closed-world* assumption to infer that all ground tuples, other than the given positive examples, are not in the relation. This implies that every other combination of blocks is not an arch. For example, blocks a1, b4, and c2 do not form an arch in the positions specified by the other six relations.

The remaining relations specify background knowledge about the position and shape of the twelve blocks. The relations *supports*, *touches*, and *left-of* identify positional relationships between pairs of blocks, and *brick*, *wedge*, and *parallelepiped* describe the shape of each block. As with the target relation these are all defined extensionally.

---

4 The input, output, and options discussed in this section are for FOIL6.

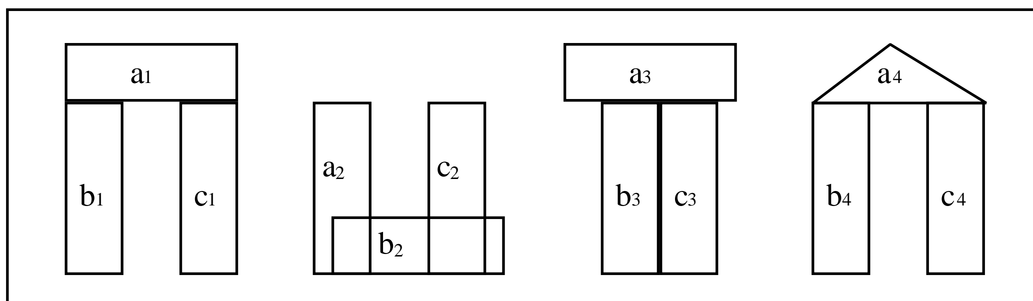


Figure 3.20 Four examples used to learn the “arch” concept.

The seven relations refer specifically to the twelve blocks in the type definition. The goal for FOIL is to find a more general description of *arch* that covers the two positive examples and any new examples described in a similar manner, though not necessarily using the same constants. If a new object comprised of three blocks a5, b5, c5 has the appropriate background relationships defined, the concept description should classify it correctly despite blocks a5, b5, and c5 not being mentioned in the training data.

To ensure the literals found by FOIL meet this criterion no constants are allowed in a clause—arguments appearing in predicates may only be variables. Some of these variables will originate from the target predicate in the head of the clause, and, within some restrictions, as many new variables as necessary may be introduced to satisfactorily complete the clause.

Restricting the search space

The search space resulting from even a small number of examples and background relations is often very large. However, much of it is either invalid in the context of the target relation, or poor in terms of classification accuracy. Determining where these regions lie in the space allows the learning algorithm to “prune” the search, resulting in more rapid completion of the learning process, with a more satisfactory concept description.

FOIL restricts the search by placing three limitations on literals that may appear in clauses. First, each literal must contain at least one variable that already exists in the clause. Thus, an n-ary predicate symbol can introduce no more than $n - 1$ new variables. This prevents the algorithm adding predicates that provide no new understanding of the target relation—a situation that may arise when the quality of the literals remaining in the search space is very poor. Another justification for this restriction concerns the order of literals in the body of the clause. Although the logical meaning of a clause is

independent of the order of literals, computational efficiency often is not. For example, the two clauses

Q(X) = R(X,Y), S(Y)

Q(X) = S(Y), R(X,Y)

have the same logical meaning, but one may be faster to compute. If *S* is easily proved to be false and *R* requires much computation, the value for *Q* will be more quickly computed by the second definition. Additionally, negated literals that contain variables not appearing in the head of the clause can change the logical meaning of the clause, depending on their position in the clause. The clause

Q(X) R(X,Y), S(Y)

is true when the relation *R* contains a tuple with a value for *Y* that does not appear in *S*. However, the clause

Q(X) S(Y), R(X,Y)

is never true if *S* contains any tuples at all. Logically the bodies of the two clauses are interpreted as *Y (R(X,Y) S(Y))* and *(Y S(Y)) Y R(X,Y)* respectively. The second interpretation has a very different meaning to the first.

The second limitation restricts the possible arguments of a predicate in the body of the clause if it is the same as the target relation, and prevents FOIL from generating clauses that are infinitely recursive. To do this FOIL identifies an *irreflexive partial ordering*, $<$, on variables in the body of the clause, such that $x < x$ never holds for any constant x. If the target predicate $P(X1, X2, ..., Xk)$ invokes the predicate $P(V1, V2, ..., Vk)$ it is sufficient for one of the pairs of variables $X1,V1$, $X2,V2$, ..., Xk,Vk to satisfy the partial ordering. For example, the second clause of the *after* relation in Figure 3.13 contains a recursive call:

after (X,Y) successor (X,Z), after (Z,Y)

To guarantee that the call does not use the value of *X* for *Z,* resulting in an infinite loop, FOIL must ensure that *X* and *Z* satisfy the partial ordering $X < Z$. The definition of the relation *successor* provides this partial ordering because if *successor(X,Z)* is true for some instantiation of *X* and *Z* there is no definition for *successor(Z,X).* By the closed-world assumption all possible definitions for *successor* that are not defined are assumed to be false, so *successor(X,Z)* is logically equivalent to $X < Z$. Conversely, the literal *after(Y,X)* cannot be considered because no partial order has been determined for *X* and *Y*.

The final limitation on the search space is a result of the heuristic used to evaluate literals. The form of this heuristic allows FOIL to disregard literals that add nothing to the understanding of the target predicate.

Evaluating literals

FOIL's basis for evaluating literals is taken from Quinlan's ID3 decision tree learner. The heuristic assesses the usefulness of a new literal by estimating the amount of information it provides in distinguishing examples of each class. Suppose a training set *Ti* contains examples of two classes, *Tin* of class *n* and *Tip* of class *p*. The information *I* required to indicate that an example in *Ti* is of class *p* is

$$I(T_i) = -\log_2 \frac{T_i^p}{T_i^p + T_i^n} \quad .$$

If a literal *L* yields the new set *Ti+1* the information to give the same signal is

$$I(T_{i+1}) = -\log_2 \frac{T_{i+1}^p}{T_{i+1}^p + T_{i+1}^n} \quad .$$

If *K* of the examples of class *p* from *Ti* are present in *Ti+1* the *information gain* regarding examples of class *p* is given by

$$Gain(L) = K \times \left(I(T_i) - I(T_{i+1})\right).$$

The gain function favours literals that select many positive examples while reducing the total number of examples in the new set. The literal that provides the largest information gain is selected for addition to the current clause.

As mentioned earlier a side effect of the *Gain* heuristic is its capacity to reduce the search space. Suppose a literal *L* contains new variables and their replacement with existing variables can never increase *K,* and such replacement at best produces a set *Ti+1* containing only examples of class *p*. If the gain achieved by this best substitution is less than the best gain achieved by any literal so far, there is no need to investigate any literals resulting from the same substitution.

Learning the concept of an arch

The arches problem provides a simple illustration of an empirical ILP system in operation. The input file in Figure 3.19 defines the relations that FOIL may use to formulate a more general definition for the *arch* relation. However, for best performance the two definitions of negative examples of the *arch* relation are excluded. If these examples are included the number of negative examples is limited to two. FOIL assumes it is being supplied with all the information necessary to learn the target relation, and the absence of other negative examples, such as a1, b4, c2, causes problems when the algorithm tries to use relations containing these values. The two negative examples in

Figure 3.20 are intended for Winston's learning system which requires "near misses" to help refine its definition of an arch (Gennari *et al*., 1990).

Omitting the near misses FOIL can assume that all other combinations of blocks are negative examples. Figure 3.21 shows the output from FOIL6, with the definition for *arch* highlighted. This definition reflects the general-to-specific nature of the search with the literals selected being minimally sufficient to distinguish the examples from those not in the relation.

4.

```
FOIL 6.2   [September 1994]
--------

Relation arch

Relation *leftof

Relation *supports

Relation *touches

Relation *brick

Relation *wedge

Relation *parallelpiped

----------
arch:

State (2/1728, 34.4 bits available)

        Save supports(B,A) (2,72 value 9.1)
        Save supports(C,A) (2,72 value 9.1)

Best literal leftof(B,C) (7.4 bits)

State (2/48, 27.0 bits available)

        Save supports(C,A) (2,3 value 17.5)
        Save touches(A,B) (2,5 value 16.3)
        Save touches(A,C) (2,5 value 16.3)

Best literal supports(B,A) (7.4 bits)

State (2/3, 20.6 bits available)

        Save clause ending with not(touches(B,C)) (cover 2, accuracy 100%)

Best literal not(touches(B,C)) (6.4 bits)

Clause 0: arch(A,B,C) :- leftof(B,C), supports(B,A), not(touches(B,C)).

arch(A,B,C) :- leftof(B,C), supports(B,A), not(touches(B,C)).

Time 0.4 secs
```

Figure 3.21 Example FOIL output for the "arches" problem.

Two programs were designed and implemented for this project. The first, HENRY, is an implementation of the INDUCT(RDR) algorithm extended to learn with relational rules[5]. The first two sections of this chapter describe HENRY, its input and output formats, extensions allowing learning of relational rules, and restrictions placed on these extensions. The final section describes ABE, a program that evaluates the ripple-down rules generated by HENRY on new examples.

4.1 HENRY

HENRY is written in C and uses the ARFF data manipulation library from the WEKA project (Holmes *et al*., 1994). This library provides support for the ARFF file format, including data structures and procedures for handling training examples. The names and legal values of attributes are read from the ARFF file, and stored for output purposes. The examples in the ARFF file are stored in a table for quick reference using array accesses.

HENRY implements both the INDUCT(RDR) and INDUCT(DNF) algorithms. The DNF version is included primarily for testing and debugging of the `best_clause` function. It is similar to the PRISM and INDUCT(DNF) covering algorithms, with some modifications to handle missing values and numeric data. Unless otherwise indicated, all references to HENRY will describe the RDR version.

Implementation overview

HENRY has four major sections. The first parses command line parameters to identify training data and relation files, determine the manner of output required, and adjust the learning algorithm to the user's needs. Table 4.1 gives a brief description of the parameters currently supported by HENRY. All but the first parameter are optional, and if none are given HENRY displays a help message.

The second section loads the training data and relation definitions into memory. Attributes are stored as an array of records, each one containing the name, data type, and an array listing the values appearing in the training data. For symbolic data this list is obtained from the attribute declaration in the ARFF file. For numeric attributes every unique value appearing in the data set is stored in a sorted list—the reason for this is explained below. The examples are stored in a table that identifies attributes by their

5 The name "HENRY" is intended to retain a link, if somewhat vague, with the program's propositional ancestor—the *henry* is the unit of electrical inductance.

Parameter	Description
–i arff file	Specifies the training data file (required).
–b	Use the binomial approximation instead of the hypergeometric function.
–c n	Use column *n* as the class value instead of the last column.
–d filename	Specifies a file to contain a DOTTY-interpretable concept description.
–D	Generate a DNF concept description instead of RDRs.
–h	Display a counter showing how much of the training data has been covered by the concept description.
–o filename	Specifies a file to contain an ABE-interpretable concept description.
–r filename | builtin | noprop	Use relational terms in the concept description—*filename* specifies a file containing relation definitions; *builtin* specifies the use of the three standard relations; *noprop* specifies that propositional terms should not be used.
–s	Disable all output normally sent to the terminal.
–t n	Terminate after *n* minutes, regardless of the state of computation, and output the current concept description.
–w	Disable warnings normally sent to the terminal.
–X ...	Available for experimental use.

Table 4.1HENRY's command line parameters.

position in the attribute list, and values by their position in the array that accompanies the attribute. Relations are stored in a linked list containing tables for each one's set of definitions.

The third section executes the learning algorithm with the parameters specified by the user. HENRY differs from inductive logic programming systems and propositional learning algorithms because it learns clauses that can contain both propositional and relational terms—the user can specify that rules contain either one, or both, of these. Because HENRY is based on the INDUCT algorithm, and INDUCT is a propositional learner, this was the first part implemented. If the user supplies no background relations, and does not specify the use of standard relations, HENRY will simply function as the INDUCT(RDR) algorithm. The next section gives a brief description of HENRY's version of the `best_clause` function in its propositional guise. The relational version is discussed in the subsequent section.

INDUCT and PRISM learn from symbolic data and only use the equality relation. Numeric attributes can be expressed in symbolic form by partitioning the range of values and assigning a symbol to each, but the lack of inequality relations, such as <, means that the order of the partitions cannot be used. HENRY uses inequality relations, and can learn

from numeric data without resorting to arbitrary partitioning. The list of unique values stored for each attribute provides a finite set of "break points" that can be evaluated in a term using either < or . In fact, HENRY uses a value midway between each consecutive pair of values as the breakpoint. This ensures that all possible partitions are represented by at least one example, and each value in the training data falls comfortably inside a partition. The training data is expected to be representative of the concept domain, so placing the breakpoint at a value specified in the data could unnecessarily segregate examples of the same class.

HENRY's fourth section displays the resulting concept description for the user. Section 4.2 discusses the three types of output produced by an in-order traversal of the RDR structure stored in memory, with specific formatting determined by the intended use of each style. HENRY also displays other information during and after the learning process. A "heartbeat monitor" indicates the percentage of examples so far covered by the concept description, and the learning time and the overall execution time are shown after each run.

Learning propositional terms

The `best_clause` function used by HENRY is shown in Figure 4.1. The vague description of the choice of the best term in Figure 2.10 has been replaced with pseudocode of the actual implementation. This part of the function can be implemented in a number of ways. The method shown in Figure 4.1 does not explicitly differentiate between numeric and symbolic attributes except at the point where relations are selected. Only the `matches` function is concerned with the actual meaning of the relations. This function determines values for z, s, k, and n (from Section 2.2), and returns the m-value for these numbers. All four values depend on the training subset passed to `best_clause`, and z and s also depend on the term being evaluated. The `matches` function runs through the training subset counting the number of examples that match the term (s), and the number of these that are positive examples (z). For propositional terms this is a matter of determining whether the specified relationship between the example's and the term's value holds. If the example's value is missing it is assumed to match the term, but z is not incremented if it is a positive example. This is the most conservative method for handling msising values. Having determined the appropriate values for z, s, k, and n, they are passed to the m-function. Depending on the parameters specified by the user this will either be the hypergeometric version or the binomial approximation.

```
function best_clause(Class_value, Attrs, Training_set): dnf_clause
var     ...
        A: attribute
        R: relation
        V: value
begin
        loop forever begin
               for A  Attrs do
                      Temp_term.A := A;
                      for R  {<,,=} such that R is applicable to A do
                           Temp_term.R := R;
                           for V  {all values of A in data set} do
                                 Temp_term.V := V;
                                 if matches(Temp_term)<matches(Best_term)
then
                                       Best_term := Temp_term
                           done
                      done
               done
               ...
        end loop
        return Clause
end best_clause
```

Figure 4.1 Pseudocode of HENRY's `best_clause` function.

In its current form `best_clause` contains loops nested three-deep. This is inefficient when two relations selected in the second loop use the same set of values because the set is rebuilt for each iteration. The < and operators apply to the same set of discrete ranges for each attribute. Combining the two relations in a single iteration would remove one iteration from the second loop—a reduction of one third. One solution is to treat the operator as *not* <, and evaluate the terms *A* < *V* and *A not* < *V* together.

HENRY's `best_clause` function, like those of INDUCT and PRISM, continues to add terms until a rule covers examples of a single class. If the training data contains many missing values, or is inadequate to distinguish classes, the algorithm may be unable to complete a rule. In such situations HENRY stops the rule at that point, warns the user of the problem, and continues with the next rule. The *if-true* branch for this node will contain examples of several classes, and the recursive call on this set will try to generate a new rule. However, HENRY will be unable to find a term to start the rule, and the branch will remain a leaf with the class of its parent.

Learning relations

HENRY can include both standard relations (<, , and =) and user-defined relations in terms. Use of standard relations is specified by a command line flag, and is applicable to any data file intended for propositional learning. In addition to searching for propositional terms to fit the data the algorithm will examine relationships between all pairs of numeric attributes, and all pairs of symbolic attributes. HENRY does not currently allow the user to specify the attributes that are comparable, and those that are not. A

```
Reading data file:
Relation: shapes2
Attributes: 4
Examples: 8

Creating rules for attribute 4: class
class value 1 - lying: 4 examples
class value 2 - standing: 4 examples

Ripple-down rules for classifying attribute 4: class
Generated from ARFF file: /home/jlittin/592/src/shapes2.arff

       IS width is-smaller-than height [4/4] f [4/4 1.429% (1.428571e-02)]?
       No     Yes -> class = standing
       |
       class = lying

User defined relations:

Number of rules: 1
Average rule length: 1.00 terms

Training time: 0.00 seconds
Total elapsed time: 0.06 seconds
```

Figure 4.2 Example output for HENRY using relational terms on the “standing up” data.

means of doing this is necessary to reduce the search space, thereby producing a quicker search, and reducing the likelihood of relationships appearing by chance. An example HENRY run using only the three standard relations on the “standing up” data set from Chapter 3 is shown in Figure 4.2. Because this is a small data set the RDR tree produced is very similar to the single DNF rule generated for each class (Figure 3.5). But by using the “lying” class as the default value HENRY does not need to generate a rule for that value—any examples not matching the rule for “standing” will be classified with the default value.

The modifications necessary for the `best_clause` function to search for these three relationships are limited to the addition of another loop at the third level (Figure 4.3). At this point in the algorithm an attribute *A* and a relation *R* have been selected. Before proceeding through the set of values for *A* in search of a propositional term the attribute is compared with every other attribute of the same type using the attribute comparison version of *R*.

User-defined relations are handled quite differently. As noted in Chapter 3 user-defined relations differ slightly from the three standard relations, and require a new recursive function, `best_rel`. This function tests every possible combination of legal values for the attributes compared by each of the user's relations. A description of the function is given in the next section.

The `best_clause` function also requires another loop shown in bold in Figure 4.4. This loop is separate from the existing search code for two reasons. First, the terms are concerned with relations specified by the user, not those that *may* exist in the data. This restricts the search to the list of extensional definitions provided by the user. The second reason is to let the user control the type of search used on a data set. Two distinct search loops provide an easy means of selecting the style of relational terms to examine.

Three restrictions are applied to user-defined relations—the first two for practical reasons due to the time limits of this project; the third results from the structure of the concept description. The first restriction is that relations must be defined extensionally. Implementation of a resolution system that could utilise intentional definitions, and verify that the definitions are not problematic, would require considerable effort. In any

```
function best_clause(Class_value, Attrs, Training_set): dnf_clause
var     ...
        A: attribute
        R: relation
        V: value
begin
        loop forever begin
                for A  Attrs do
                        Temp_term.A := A
                        for R  {<,,=} such that R is applicable to A do
                                Temp_term.R := R
                                for A'  Attrs, A'  A, A' is comparable to A
do
                                        Temp_term.A' := A'
                                        if m(Temp_term) < m(Best_term) then
                                                Best_term := Temp_term
                                done
                                for V  {all values of A in data set} do
                                ...
                        done
                done
                ...
        end loop
        return Clause
end best_clause
```

Figure 4.3 Pseudocode outline of `best_clause` used in the relational extension to HENRY.

case, this limitation is also enforced by FOIL6, so no ground is yielded here when comparing the algorithms.

Unlike FOIL, which can use the closed-world assumption to infer the class of all other possible examples, HENRY requires negative examples of a concept. Like most other propositional learners, HENRY expects the training data to be representative of the distribution of classes in the whole population. If no negative examples are presented the algorithm assumes that no such examples exist, and generates a concept description that says all instances are positive. To simulate the closed-world assumption it is necessary to include every possible negative example in the training data file. This is not difficult in principle, although there may be many such examples.

The third restriction imposed by HENRY is a result of the structure of ripple-down rules. Most ILP systems allow recursive definitions of the target predicate to be generated. These definitions are usually composed of several clauses, with one providing a base for the recursive call of the other. The target predicate in ripple-down rules is defined by a single RDR tree, and HENRY does not allow terms to “call” the top level node of the tree. This would be akin to having a recursive logic program clause that could only refer to itself. Such a definition would never terminate with a true result—only the failure of a term would cause computation of the clause to cease.

Variables

The `best_rel` function introduces variables into the relation currently being evaluated. The function is tail recursive, allowing it to cycle through all possible constant values while introducing new and previously used variables into the relation. Unlike FOIL, HENRY can use constants in relational terms. Although a term containing only constants, such as *left_of (a1, b1)* would match every example because it is always true by definition,

```
function best_clause(Class_value, Attrs, Training_set): dnf_clause
...
begin
	loop forever begin
		for A  Attrs do
			...
		done
		for R  {user-defined relations} do
			for Rdef  {definitions of R} do
				Temp_term := best_rel(R,Rdef)
				if m(Temp_term) < m(Best_term) then
					Best_term := Temp_term
			done
		done
		...
	end loop
	return Clause
end best_clause
```

Figure 4.4 Pseudocode outline of the `best_clause` function used to search the space of user-defined relations.

combinations of variables and constants might prove useful. For example, a term such as *left_of (X, b1)* may be fruitful in a problem classifying combinations of blocks where individual blocks may be included in several objects. Any object containing a block that is to the left of block *b1* will match this term.

HENRY will try introducing between 1 and $n - 1$ variables into an n-ary relation. If this relation is the first to appear in the clause all variables will be new, and will bind to every value in the same position of the relation's definition. For example, the variable *X* in the term *left_of (X, Y)* will bind to the set {*b1, b2, b3, b4*} from the definition in Figure 3.19. HENRY does not try using the same variable more than once in any term, because of uncertainty about the bindings of each argument in logical and computational terms.

If relational terms already exist in the current clause HENRY examines all valid combinations of the variables in these terms before introducing new ones. For example, if the clause already contained the term *left_of (X, Y),* HENRY would try the variables *X* and *Y* in a new relation. Using the variable *X* in the term *supports (X, Z)* (from Figure 3.19) reduces the set of possible bindings for *X* to {*b3, b4*}. If a new variable had been introduced here it would be bound to {*b1, b3, b4, c1, c3, c4*}. HENRY's use of existing variables where possible is in line with FOIL's policy on the matter. Although HENRY does not require an existing variable to be used in any new relational term, in practise this usually appears to happen (see examples in Section 5.2).

HENRY does not place many restrictions on the search space with regard to using variables. None of the three restrictions discussed in the description of FOIL are implemented in the `best_rel` function. Preventing problematic recursion is perhaps the most important of these three, but because ripple-down rules cannot reference themselves this issue does not arise. However, restrictions on the introduction of new variables would make the search more efficient.

4.2 Ripple-down rule output formats

HENRY presents ripple-down rules in three formats. The first is intended for user interpretation, and is a modified form of that presented in Chapter 2; the second is for an evaluation program to test on new examples; the third is used by a directed graph drawing package to present the ripple-down rules in a format suitable for use in paper documents. All three are generated by a recursive procedure that traverses the RDR data structure in memory.

Text-based format

The default output style is a text-based binary tree (Figure 4.5) that is printed to the user's terminal when the program terminates. The structure differs from that shown in Chapter 2 in that the default classification, formerly at the top of an RDR substructure, has been moved to replace the empty *if-false* node at the bottom of the structure. The change retains the meaning of the original RDR structure, and makes it easier to read because the user no longer has to keep track of default classifications from higher up in the structure. This representation also eliminates the need for the null rule in the top-most node.

The text-based tree in Figure 4.5 was generated from the iris data, and provides a good comparison with the disjunctive normal form rule set of the same data in Figure 2.1. This format is intended for human use, and provides more information than the machine-readable Lisp version described next. The first line of the structure is a rule that would appear in the second box of the RDR structure in Chapter 2. This rule is of the *if... then... else...* type, as described in Section 2.1, but it is presented as a question that is either true or false with respect to an example being classified. If the example matches the rule the *Yes* branch of the tree is followed, otherwise the *No* branch is followed. The structure is traversed as described in Chapter 2, except that a classification is given at the final *if-false* (*No*) node in the structure as described earlier.

The other information spread throughout the rule is largely for the user's interest. The square-bracketed ratio after each term shows the number of positive examples selected at this point in the rule's execution, over the number of examples of all classes that it selects—this is the same *z/s* ratio used by PRISM. The letter following the ratio,

```
  IS "petallength" >= 2.45 [50/100] t AND "petalwidth" < 1.75 [49/54] t AND
"petallength" < 5.35 [49/52] t [49/52 0.000% (1.076637e-34)]  ?
  No      Yes
  |       |
  |       IS "petallength" >= 4.95 [2/4] f AND "petalwidth" < 1.55 [2/2] t [2/2
10.962% (1.096197e-01)]  ?
  |       No      Yes -> "class" = "Iris-virginica"
  |       |
  |       IS "sepallength" < 4.95 [1/2] f AND "sepalwidth" >= 2.45 [1/1] t [1/1
33.333% (3.333333e-01)]  ?
  |       No      Yes -> "class" = "Iris-virginica"
  |       |
  |       "class" = "Iris-versicolor"
  |
  IS "petallength" >= 3.35 [47/48] t [47/48 0.000% (4.097019e-34)]  ?
  No      Yes
  |       |
  |       IS "petallength" < 4.85 [1/3] t AND "sepallength" < 5.95 [1/1] t [1/1
33.333% (3.333333e-01)]  ?
  |       No      Yes -> "class" = "Iris-versicolor"
  |       |
  |       "class" = "Iris-virginica"
  |
  "class" = "Iris-setosa"
```

Figure 4.5 Text-based ripple-down rule tree generated from the iris data.

```
( setf ruleset
'((rule "petallength" >= 2.45 t AND "petalwidth" < 1.75 t AND "petallength" < 5.35
t)
	((rule "petallength" >= 4.95 f AND "petalwidth" < 1.55 t)
		(classification "class" "Iris-virginica")
		((rule "sepallength" < 4.95 f AND "sepalwidth" >= 2.45 t)
			(classification "class" "Iris-virginica")
			(classification "class" "Iris-versicolor")))
	((rule "petallength" >= 3.35 t)
		((rule "petallength" < 4.85 t AND "sepallength" < 5.95 t)
			(classification "class" "Iris-versicolor")
			(classification "class" "Iris-virginica"))
		(classification "class" "Iris-setosa"))))

(setf user-defined-relations nil)
```

Figure 4.6 Lisp-based ripple-down rule tree generated from the iris data.

either a *t* or an *f*, is part of an experiment in dealing with missing values (see Section 5.1). The letter indicates whether or not an example with a missing value for the attribute in the preceding term should match the term (*t*) or not (*f*). The value expressed is the result of matching a hypothetical example that has the attribute's average value for the training examples that remained at that point in the rule's construction. This value is also present in the machine-readable format of the RDR structure, and its use in classification depends on the evaluator. Normally an example fails to match a term if the value for the specified attribute is missing.

The final square-bracketed field in the rule contains information about the entire rule. The first item is the *z/s* ratio for the entire rule, and is the same as the value for the last term. The next item is the *m*-value (expressed as a percentage) for the entire rule, and the final item is the *m*-value expressed as accurately as possible in floating point form. This information is provided to differentiate *m*-values that are very similar, particularly those close to 0, such as the first rule and its *No* descendent in Figure 4.5 which are both 10–34. The high precision allows HENRY to choose between terms with *m*-values that are this similar. The percentage value given prior is intended for "at a glance" comparison of rules.

Lines in the RDR tree that do not begin with *IS* are classification nodes. When an example works its way down the tree to one of these nodes it is classified with the label at the node. An example that matches the first rule in Figure 4.5 but neither of the rules in its *Yes* branch will be classified as "Iris-versicolor" by the leaf node in the *No* branch of the third rule. In the RDR structure of Chapter 2 this classification would be made with the default value of the second rule.

Lisp format

The second output format is used by an external evaluation program, ABE, described in Section 4.3. ABE is written in Common Lisp so this format presents ripple-down rules as

a Lisp data structure (Figure 4.6). The structure is similar to the text-based format of the previous section in that the concept description is represented as a binary tree. Most Lisp data structures are constructed from lists of objects, and the binary tree is formed from a set of nested lists. A Lisp binary tree has one of two forms:

(rule-list if-true-subtree if-false-subtree),

or

(classification class-name class-value).

The *rule-list* item at the head of the first list represents the conjunction of terms to be evaluated at this point in the tree, and *if-true-subtree* and *if-false-subtree* are binary trees. If an example matches the rule the *if-true-subtree* is traversed, otherwise the *if-false-subtree* is traversed. The second form represents a leaf node, and provides a classification for examples reaching that node.

The *rule-list* item is itself a list containing the conjunction of terms to be matched at the node. For example:

(rule "petallength" >= 4.95 f AND "petalwidth" < 1.55 t).

The first item, *rule,* lets ABE recognise this as a rule, rather than a leaf node. The remainder of the list is a series of propositional terms separated by *AND* items. The *t*s and *f*s at the end of each term match those in the text-based format described earlier.

The last line of the Lisp-based output declares a variable called *user-defined-relations.* This is part of the relational learning system and is discussed in Section 4.3.

Dotty format

The final output format is also intended for interpretation by an external program—in this case DOTTY, an application for drawing directed graphs. DOTTY reads a file containing a definition of a graph, and formats the graph nicely on the screen. The binary-tree structure of ripple-down rules is suited to representation as a directed graph, and DOTTY provides a means of reproducing concept descriptions in a format suitable for inclusion in paper documents (Figure 4.7).

HENRY generates an input file for DOTTY such as the one in Figure 4.8. Each line represents either a node in the graph or an edge between two nodes. Nodes representing rules in the RDR structure are labelled with the rules, and those representing classifications (leaf nodes) with the class value. Every rule node has two edges leading to two other nodes; the leftmost edge leading to the *if-true* subtree and the rightmost edge leading to the *if-false* subtree. These edges are labelled *T* and *F*.

Translating ripple-down rules to PROLOG

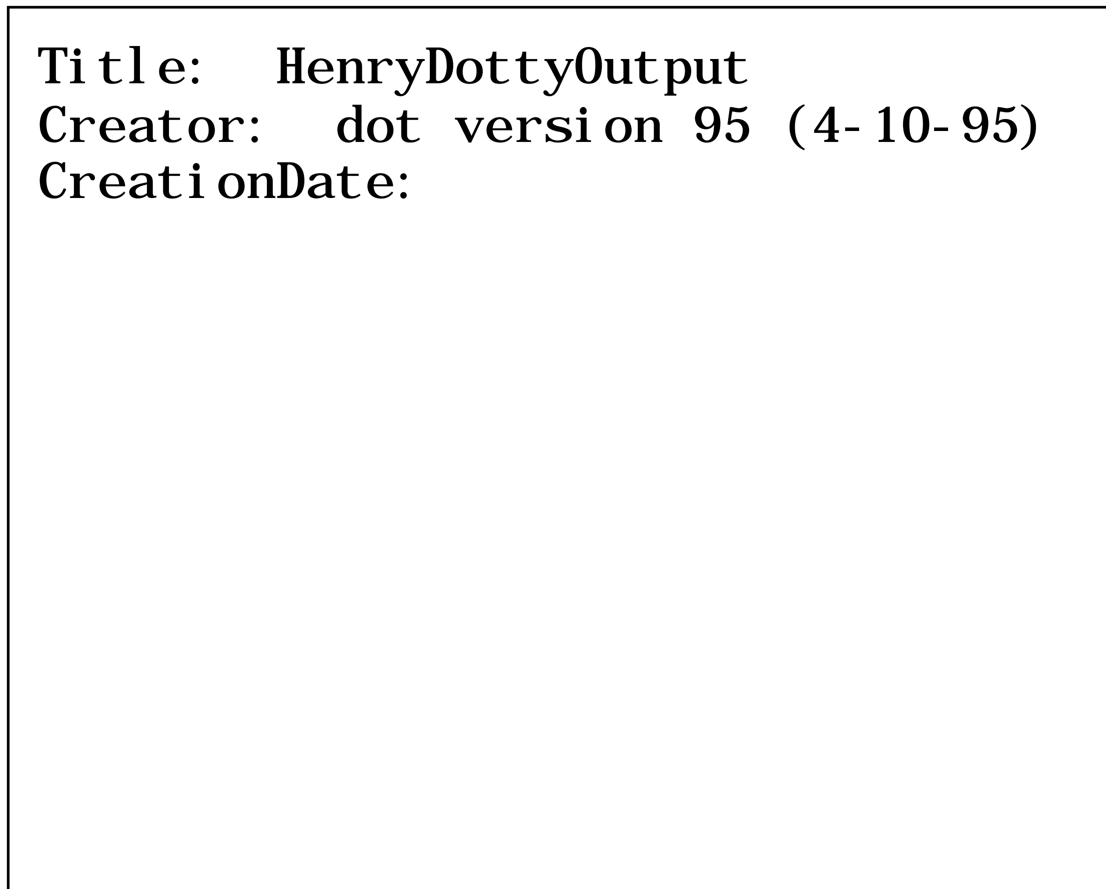


Figure 4.7 Ripple-down rules for the iris data as displayed by DOTTY.

```
digraph HenryDottyOutput {
node_1 [label="petallength >= 2.45\npetalwidth < 1.75\npetallength < 5.35"]
node_2 [label="petallength >= 4.95\npetalwidth < 1.55"]
node_3 [color=lightgray style=filled shape=box label="Iris-virginica"]
node_2->node_3 [label="T"]
node_4 [label="sepallength < 4.95\nsepalwidth >= 2.45"]
node_5 [color=lightgray style=filled shape=box label="Iris-virginica"]
node_4->node_5 [label="T"]
node_6 [color=lightgray style=filled shape=box label="Iris-versicolor"]
node_4->node_6 [label="F"]
node_2->node_4 [label="F"]
node_1->node_2 [label="T"]
node_7 [label="petallength >= 3.35"]
node_8 [label="petallength < 4.85\nsepallength < 5.95"]
node_9 [color=lightgray style=filled shape=box label="Iris-versicolor"]
node_8->node_9 [label="T"]
node_10 [color=lightgray style=filled shape=box label="Iris-virginica"]
node_8->node_10 [label="F"]
node_7->node_8 [label="T"]
node_11 [color=lightgray style=filled shape=box label="Iris-setosa"]
node_7->node_11 [label="F"]
node_1->node_7 [label="F"]
}
```

Figure 4.8 DOTTY input file for the ripple-down rules generated from the iris data.

Although the ripple-down rule structure has characteristics that make it different from the logic programs generated by FOIL, it is possible to translate RDRs to a PROLOG-like DNF form. The procedure is identical to that of producing DNF rules from an ID3 or C4.5 decision tree. A clause is produced for each leaf node by traversing the tree and adding the terms at each internal node to the clause. If an *if-false* branch is in the path to the leaf the terms at the node are bracketed, and the expression is negated. Figure 4.9 shows an example RDR tree with the path to a leaf node highlighted. This path translates to the clause shown at the bottom of the figure. A set of such rules is logically equivalent to the RDR tree they are translated from, and retains the structure's properties. For example, the rule set will not give multiple classifications which are possible with rules initially generated in DNF form.

Although the PROLOG form of an RDR tree can be viewed as a logic program HENRY is not an inductive logic programming system, because the DNF rules violate a number of the principles of ILP. For example, the negation of a conjunction of literals, such as *not (L*1,*L*2*),* is not allowed in the body of a clause. However, HENRY can duplicate in RDR form many of the concept descriptions generated by inductive logic programming systems, with the only real limitation being on recursive definitions.

4.3 **ABE**

The PROLOG/DNF form of ripple-down rules allows PROLOG-based evaluators, such as PREVAL (Garner *et al*., 1995), to evaluate them on new data sets, but the format is rather verbose, and requires duplication of many parts of the structure. An RDR-specific

```
IS "petallength" >= 2.45 AND "petalwidth" < 1.75 AND "petallength" <
5.35 ?
  No     Yes
   |      |
   |      IS "petallength" >= 4.95 AND "petalwidth" < 1.55 ?
   |      No     Yes -> "class" = "Iris-virginica"
   |      |
   |      IS "sepallength" < 4.95 AND "sepalwidth" >= 2.45 ?
   |      No     Yes -> "class" = "Iris-virginica"
   |      |
   |      "class" = "Iris-versicolor"
   |
  IS "petallength" >= 3.35 ?
  No     Yes
   |      |
   |      IS "petallength" < 4.85 AND "sepallength" < 5.95 ?
   |      No     Yes -> "class" = "Iris-versicolor"
   |      |
   |      "class" = "Iris-virginica"
   |
  "class" = "Iris-setosa"


class(Iris-virginica) :- petallength >= 2.45, petalwidth < 1.75,
       petallength < 5.35, not (petallength >= 4.95, petalwidth < 1.55),
       sepallength < 4.95, sepalwidth >2.45.
```

Figure 4.9 Translation of ripple-down rules to an equivalent PROLOG-like format.

evaluator requires little change to the standard output format, and it can evolve as necessary to contend with changes to the rule formats—a point endorsed by the introduction of relational terms in the rules.

ABE (*A Basic Evaluator*) is written in Common Lisp and uses list-like data structures to represent rules and binary trees. The choice of Lisp as the implementation language was largely pragmatic. It provides many useful data structure manipulation facilities that are frequently employed because ABE spends most of its time searching through a binary tree structure matching lists of attributes along the way. Although HENRY must do this also, the evaluator is kept separate from the learning program. The different nature of the learning and evaluation tasks, and the distinct input and output corresponding to each, implies a different program for each task. It would be bad software engineering practise for HENRY to perform both tasks.

Input and output

The format of the input rule structure (described in Section 4.2) represents the entire RDR tree as a nested list. ABE reads this file as a normal Lisp input file and assigns the

ripple-down rules to the variable *ruleset*. A second variable, *user-defined-relations*, also specified in this file, is a list of the extensional definitions for relations appearing in the rule set. The ARFF library filter ARFFTOLISP converts the test data file into three Lisp data structures. The first contains the target relation's name; the second is a list of the concept's attributes; and the third is a list of examples, each expressed as a list of values.

ABE's output is usually a list of the examples misclassified by the rule set. Each example is identified by its place in the ARFF file, and is displayed in full. Following this, ABE displays a *confusion matrix*—a grid with the class values along both axes. Entries in the matrix show the distribution of correct and incorrect classifications for each class value. Figure 4.10 shows ABE output from a test on the iris data. The 150 examples were divided into two data sets—66% for training and 34% for testing. HENRY was run on the training sample, and the resulting RDR tree, along with the test data, was presented to ABE for evaluation. The ripple-down rules misclassified four irises in the test data, #3 and #13 of cultivar iris versicolor, and #27 and #33 of cultivar iris virginica. The confusion matrix shows that the two misclassified iris versicolor were predicted to be iris virginica, and the iris virginica were predicted to be iris versicolor. This indicates HENRY had little difficulty distinguishing iris setosa from the other two cultivars, but the examples of iris versicolor and iris virginica were sufficiently similar to produce some overlap in the classifications. This characteristic is well documented for this data set (Salzberg, 1990).

In addition to displaying misclassified examples ABE can display every example with its HENRY-predicted class value. This output can be filtered to create a new data set using the predicted class value in place of the original value.

```
%% ABE - A Better Evaluator, Version 951218.1654 %%
ABE is evaluating rulefile iris-train-rules.l on data file iris-test.arff
;; Loading file _.rules.l ...
;; Loading of file _.rules.l is finished.
;; Loading file _.arff.l ...
;; Loading of file _.arff.l is finished.

Misclassified instance #3: 5.9,3.2,4.8,1.8,"Iris-versicolor"
Misclassified instance #13: 6.2,2.2,4.5,1.5,"Iris-versicolor"
Misclassified instance #27: 6.3,2.8,5.1,1.5,"Iris-virginica"
Misclassified instance #33: 4.9,2.5,4.5,1.7,"Iris-virginica"

Correctly Classified Instances    47    92.16%
Incorrectly Classified Instances  4     7.84 %
                                 ------
                                  51

 Classified as -> Iris-se.. Iris-ve.. Iris-vi..
                  -----------------------------
      Iris-setosa 17
  Iris-versicolor           16        2
   Iris-virginica           2         14
                  -----------------------------
DONE
```

Figure 4.10 Output from ABE evaluating ripple-down rules on the iris data.

Matching examples

ABE employs a recursive function to traverse the RDR structure using the method described in Chapter 2. The test example passes down the left branch of a node if it matches the rule at that node, and down the right branch if it does not match. Reaching a leaf node, the classification given at the node is compared with the actual class value of the example, and the confusion matrix is updated accordingly.

To classify an example ABE compares it to the terms in each rule from left to right (the order they were added to the rule by HENRY). If at any point a term is encountered that does not match the example the rule fails, and the example is immediately passed down the *if-false* branch of the node. If the current term is propositional the list representing the example is searched to find the value for the term's attribute. The term's value is then examined, and the corresponding relationship between the values in the term and the example is assessed.

If the current term is relational ABE searches the list of extensional definitions of the specified relation for one that is satisfied by the example. If no such a definition is found the rule immediately fails, and the example passes down the node's *if-false* branch. For example, if the relational term

(C = 1 AND D = 1 AND left-of (C D))

was to be matched with an example with value 0 for *block_e* and 1 for attributes *block_d* and *block_f*, ABE would search the list of definitions of *left-of* for one that matched the example. The definition *(left_of block_d block_f)* would be satisfied by the example, but the definition *(left_of block_e block_f)* would fail because *block_e* and *block_f* do not both have the value 1.

Missing values

ABE can treat examples with missing values in two ways. First, and in common with most rule evaluation methods, a missing value always fails to match a term using the particular attribute. With ripple-down rules this procedure will at worst result in an example being given the default classification. A similar example classified with DNF rules will usually have no prediction made because it fails to match any rule.

The second method uses the *t*s and *f*s described in Section 4.2. If a term has a *t* for its missing value option any examples with missing values for its attribute will automatically match. Similarly, any such examples will fail to match a term with an *f* as the missing value option. This method assumes the missing value in the example is near the mean for that value (determined from the training data). If the assumed value is lower than the value specified in a term with the operator <, the missing value option would be *t* and the

example would match the term. A comparison of the two missing value methods is presented in Chapter 5.

5.

Results

This chapter presents results of eight experiments using HENRY and ABE. The first four experiments evaluate HENRY on sixteen propositional data sets, and compares it with other machine learning algorithms. The algorithm also provides the basis for comparing different term evaluation heuristics and rule evaluation methods. The last four experiments involve relational learning and test the corresponding extensions to HENRY.

5.1 Experiments on UCI data sets

The sixteen data sets originated from the UCI machine learning repository, and form a core of data sets used for many empirical comparisons of machine learning algorithms (Salzberg, 1995). The data sets and method of evaluation used here are the same as those used by Holte (1993) in his 1R experiments. The data sets range in size from 47 to 8124 examples, with between 5 and 36 attributes. A summary of the contents of each data set is given in the appendix.

Table 5.1 shows the results for eight learning schemes on the sixteen data sets. Each scheme was run twenty-five times on each data set, using a random selection of 66% of the examples as training data, and the remaining 34% as test data. The first column lists the data sets, and the second shows the percentage of values that were missing in each one. The values in the last seven columns are the average percentage accuracy on the test data over the twenty-five runs. The bottom row of the table shows the average classification accuracy for each scheme over the sixteen data sets; this gives an indication of the relative performance of each scheme.

The schemes used were (from left to right in Table 5.1) HENRY using the hypergeometric measure, HENRY using the binomial approximation, HENRY using an informational measure similar to that of ID3, C4.5 producing a pruned decision tree, 1R, HENRY with the missing value evaluation heuristic, HENRY using =,<, and inter-attribute relations, and FOIL. Unless otherwise indicated HENRY uses the hypergeometric measure for selecting terms, and concept descriptions are evaluated with missing values always failing to match terms.

	MISSING	HENRY HYPERGEOMETRIC	HENRY BINOMIAL	HENRY INFORMATIONAL	C4.5	1R	HENRY MISSING VALUES	HENRY RELATIONAL	FOIL
BC	0.31%	64.0	64.8	64.0	71.4	68.7	64.2	63.6	54.1
CH	-	98.2	98.1	93.8	99.3	67.6	98.2	98.1	29.7
G2	-	74.0	74.3	68.5	73.5	72.9	74.0	72.8	67.8
GL	-	69.1	67.6	51.5	66.9	53.8	69.1	49.1	49.6
HD	0.17%	72.3	72.2	71.3	73.0	73.4	72.2	53.9	65.1
HE	5.39%	78.7	78.5	76.4	70.1	76.3	79.8	47.5	67.3
HO	22.77%	73.9	77.3	76.2	77.3	81.0	74.7	77.0	61.3
HY	6.48%	98.6	98.7	79.6	91.0	97.2	99.0	96.8	98.0
IR	-	93.6	93.3	93.5	95.1	93.5	93.6	97.1	91.0
LA	33.64%	76.2	79.4	77.9	74.1	71.5	82.5	62.8	70.3
LY	-	77.8	78.0	72.0	74.9	70.7	77.8	73.4	64.7
MU	1.33%	100.0	100.0	100.0	100.0	98.4	100.0	100.0	99.6
SE	6.48%	96.8	96.5	78.2	75.7	95.0	96.8	92.2	95.1
SO	-	97.2	97.2	96.5	96.8	91.0	97.2	91.7	96.8
V1	5.47%	87.7	87.2	85.7	83.8	86.8	87.4	86.5	77.3
VO	5.30%	94.8	94.1	90.0	93.4	95.2	94.7	93.6	88.0
mean		84.6	84.8	79.7	82.3	80.8	85.1	78.5	73.5

Table 5.1 Accuracy results for eight learning schemes on sixteen UCI data sets using the method described by Holte (1993).

The first experiment compares the classification accuracy of three algorithms: HENRY (simulating INDUCT(RDR) with numeric extensions), C4.5, and 1R. C4.5 (Quinlan, 1993) is the benchmark by which machine learning algorithms are compared, whereas 1R (Holte, 1993) provides a baseline accuracy value for any data set. The second experiment uses HENRY to compare three term selection measures. This comparison is intended to determine the effectiveness of the binomial approximation of the hypergeometric measure, and to compare the probabilistic approach with an information-based approach. The third experiment examines the abilities of relational learners with propositional data. FOIL is compared with HENRY using the three “standard” relations introduced in Section 3.1. The final experiment compares two methods for evaluating rules on examples that have missing values. The first method fails any example with a missing value for the attribute in a term. The second method uses the technique described in Section 4.3 where a truth value for each term is generated from the average value for the particular attribute in the training data. C4.5 uses a similar technique for classifying examples with missing values, and this can increase the accuracy of a decision tree by several percent. In these experiments C4.5 decision trees are evaluated with PREVAL,

using the "missing values fail" method—the fourth experiment is intended to evaluate different methods for evaluating ripple-down rules.

Comparing propositional algorithms

The aim of this experiment was to compare the INDUCT(RDR) algorithm with two other propositional learning algorithms: C4.5 and 1R. C4.5 is a decision tree learner and uses several techniques that differ from their counterparts in HENRY. The difference in performance of the systems will reflect the combined effectiveness of these techniques. The first dissimilarity is C4.5's use of an information-based metric for assessing the utility of attributes. The attribute providing the largest information gain (as described in Section 3.3) at each node in the decision tree is used to split the tree. If the attribute is symbolic each of its values is used to create a branch down which examples with the particular value pass. If the attribute is numeric a threshold value is determined, and a binary split is used. Examples with values less than the threshold pass down one branch, and those with values greater than or equal to the threshold pass down the other. The second difference is that a C4.5 decision tree has a single attribute test at each node, and if the attribute used is symbolic the tree need not be binary. The ripple-down rule structure is binary because symbolic terms specify a single value, and instead of one attribute–threshold-value term at each node, RDRs may have a conjunction of such terms.

1R generates a rule set that uses a single attribute to divide the data set into individual classes. The rules are intended to be the simplest that can be used to categorise the data, and therefore provide a baseline accuracy for other algorithms to better. Any concept description larger than the 1R rule set, but less accurate at classifying new examples, is overfitting the training data.

Method

The three algorithms were evaluated on the sixteen data sets in the manner described by Holte (1993). ABE evaluated HENRY's ripple-down rules on the test data, using the "missing values fail" method. C4.5 decision trees were evaluated by converting to them to the equivalent DNF representation in a PROLOG-like form, and processing them with PREVAL. Accuracy values for 1R are from Holte (1993).

Results

The results pertaining to this experiment are in columns three, six, and seven of Table 5.1. HENRY was the best of the three schemes on ten of the data sets (G2, GL, HE, HY, LA, LY, MU, SE, SO, V1); C4.5 was best on four data sets (BC, CH, IR, MU); and 1R was best on three data sets (HD, HO, VO). HENRY also had the highest average accuracy of

84.6% on the test data; C4.5 had an average accuracy of 82.3%; and 1R had an average accuracy of 80.8%.

Conclusions

These results show that all three algorithms are closely matched on these sixteen data sets. HENRY is at least as accurate as C4.5 on ten of the data sets indicating that it may have a slight edge. However, as discussed by Salzberg (1995), it is not possible to say definitively that one algorithm is significantly better than the other on these particular data sets. It is possible to "tune" either algorithm to perform better on any particular data set, and the default values for various thresholds are chosen so the algorithms perform well on a range of data sets. If a different group of data sets had been chosen C4.5 may have produced slightly higher accuracy figures than HENRY. Nevertheless, there is sufficient evidence to suggest that the INDUCT(RDR) algorithm is comparable to C4.5 in terms of accuracy.

This experiment provides several insights into the UCI data sets. First, the three data sets on which 1R was the most accurate may indeed be characterised by the values of a single attribute. In the VO data set this is certainly the case—the attribute *physician-fee-freeze* can be used to correctly classify 95.6% of the examples by simply saying "*class* is democrat if *physician-fee-freeze* is false". Second, the extra complexity of the concept descriptions generated by HENRY and C4.5 provides less than 5% greater accuracy on the sixteen data sets. As discussed by Holte (1993), this is a characteristic of data sets typically used for evaluating machine learning algorithms. These data sets generally have simple target concepts that are well described by simple rule sets, such as those of 1R.

Comparing rule evaluation heuristics

The aim of this experiment was to evaluate term selection heuristics. The hypergeometric measure is compared with the binomial approximation and a version of the information-based metric used in C4.5 and FOIL. The hypergeometric formula, used to estimate the probability that a rule can be bettered by a random selection of examples, is expensive to compute. With many examples, values used in the computation can approach the bounds of the processor's numeric precision. Therefore, the use of a less expensive approximation would be beneficial in terms of speed and precision. The binomial approximation and the informational metrics are considerably less costly than the hypergeometric measure. Although the binomial measure is intended as a replacement for the hypergeometric measure, the informational measure is evaluated here as an alternative approach to assessing the "quality" of a rule.

Method

The three rule evaluation heuristics are all implemented in HENRY—a command line parameter indicates which measure to use when selecting terms—otherwise, the algorithms are identical for each run. Again the experiment used the sixteen UCI data sets in the manner described by Holte. ABE evaluated the ripple-down rules produced by each heuristic, using the “missing values fail” method.

Results

The results for this experiment are in columns three, four, and five of Table 5.1. The average accuracy values across the sixteen data sets for the hypergeometric and binomial metrics are similar at 84.6%, and 84.8%, respectively. The informational metric has a slightly lower average accuracy of 79.7%. The average size of the ripple-down rule structure produced by the hypergeometric and binomial measures was also similar at 16.0 rules (internal nodes) with 2.8 terms per rule, and 16.7 rules with 2.9 terms per rule, respectively. The informational measure produced average RDRs with 80.4 rules, and 3.0 terms per rule.

Conclusions

The average accuracy and RDR structure size of the hypergeometric and binomial measures are almost identical, indicating the less expensive binomial approximation is a worthy replacement for the hypergeometric measure. The informational measure was nearly as accurate as the two probabilistic measures, but its concept descriptions were far larger. Although there is no obvious reason for this, it is worth noting as FOIL, the relational algorithm used in the next experiment, also uses an information-based search measure.

Comparing relational algorithms

The aim of this experiment was to compare the performance of HENRY and FOIL on propositional data. FOIL was designed primarily to operate on relational data, but it is capable of generating rules using propositional terms. Although these rules do not conform to the restrictions imposed by inductive logic programming, they are necessary when no background knowledge is provided. HENRY is able to search for both propositional and relational terms in propositional data using the three standard relations =, <, and . When searching the space of new literals HENRY evaluates relational terms first. A “tie” between a propositional term and a relational term will always be resolved in favour of the relational term, because a new term must be significantly better than the existing best term to replace it.

Method

Again HENRY and FOIL were evaluated in the manner described by Holte. HENRY's RDRs were evaluated with ABE, using the "missing values fail" method. Because FOIL generates a concept description that classifies an example as being either a member or not a member of a relation, it was necessary to create a training set for every class value present in each data set. For a three-class problem FOIL would be run three times, with each class once being the target concept, and twice being part of the set of negative examples of another class value. The three rule sets were then combined to produce a single set that covered every class. PREVAL was used to test the rule sets generated by FOIL.

Results

The results for this experiment are in columns nine and ten of Table 5.1. The average accuracy values of the two algorithms are similar at 78.5% for HENRY and 73.5% for FOIL. The individual values are quite different for many of the data sets, with the most extreme difference being for the CH data set where HENRY scores 98.1% and FOIL scores 29.7%—a difference of 68.4%.

Conclusions

As with the previous experiments the difference in average accuracy is too small to indicate a decided advantage for HENRY. However, it is noteworthy that HENRY continues to come out on top in these close situations. The most obvious point of interest is the often large difference in accuracy on some data sets, particularly CH. In most cases HENRY's results are close to those for the purely propositional runs. Figure 5.1 shows a section of the relational RDR tree generated from the IR data set. The profusion of propositional terms suggests the concept is best described in that form. The lower accuracy of the relational version indicates that, by searching the relational terms before propositional terms, HENRY may be making poor decisions where the m-values for two terms are very similar. The relational terms selected may be identifying regularities in the training data that are better expressed with propositional terms. Swapping the order of search, and thus specifying a preference for propositional terms, could result in better accuracy on these data sets. However, the existence of relationships within the data that are truly better expressed with relational terms would obviate such "tuning" of the algorithm.

FOIL's poor performance on many of the data sets indicates it is less suited to classification tasks of this form. Whereas HENRY can combine relational and propositional terms as necessary, FOIL uses either one or the other. Given purely propositional data, even when represented in relational form, FOIL acts as a propositional learner, producing PROLOG-like rules that contain propositional literals. The poor

```
  IS sepalwidth is-bigger-than petallength [31/31] f [31/31 0.000% (2.185124e-26)]
?
  No      Yes -> class = Iris-setosa
  |
  IS petalwidth < 1.75 [32/35] t AND petallength < 5.35 [32/33] t [32/33 0.000%
(1.103604e-23)]  ?
  No      Yes
  |       |
  |       IS petallength > 5.05 [1/2] f AND sepallength >= 6.15 [1/1] f [1/1 35.354
% (33.535354e-01)]  ?
  |       No   Yes -> class = Iris-virginica
  |       |
  |       class = Iris-versicolor
  |
  IS petallength < 4.85 [1/3] t AND sepallength < 5.95 [1/1] t [1/1 33.333%
(3.333333e-01)] ??
```

Figure 5.1 A section of a relational ripple-down rule tree generated from the IR data set.

performance on some of these data sets may be a reflection of the performance of the informational measure used in the previous experiment.

The relative unsuitability of relational learning to these data sets again illustrates Holte's point that these "standard" data sets contain very simple concepts.

Comparing missing value methods

The aim of this experiment was to evaluate different methods for dealing with missing values in ripple-down rules. The normal method is to fail the match of any example that has a missing value for an attribute specified in any term of a rule. In DNF rule sets the worst case scenario for such a procedure results in an example failing every rule and being unclassified. Ripple-down rules do not allow this to happen because the example would reach the leaf of the RDR tree that specifies the original default classification. Thus, the example would be classified as the most frequent class in the training data.

The alternative method employed in this experiment appends a truth value to each term in the RDR tree that specifies whether a missing value should match or fail to match the term. The truth values are determined by assuming the example has the average value of the examples in the training data for the attribute in question. If the attribute is symbolic this value will be the most frequent in the training data.

When ABE is evaluating the ripple-down rules on a test example and a missing value is encountered, the program examines the truth value associated with the current term and matches or fails the example on the strength of that value. Thus, an example with all values missing would be considered an instance of the most representative examples of the concept. All the examples at any node in the RDR tree share a number of features that allowed them to reach the node. This approach assumes any of these examples with a missing value will also share that value with the other examples at the node. This experiment will determine if the difference in approach is significant.

Method

HENRY was run twice on the sixteen data sets in the manner described by Holte. On the first run ABE used the “missing values fail” method, and the “average value” method on the second run. Note that the concept descriptions generated by in both runs would be identical for the same training data—it is the way they are used to classify new examples that is different.

Results

The results for this experiment are in columns three and eight of Table 5.1. As expected the results are identical for data sets with no missing values (indicated by dashes in the MISSING column of Table 5.1). The average accuracy is 84.6% for the first method, and 85.1% for the second. Table 5.2 shows the results on the seven data sets with more than 5% of the values missing. On these data sets the average accuracy is 86.7% for the first method, and 87.8% for the second.

Conclusions

The average accuracy of the two evaluation methods is very similar with the “average value” method having a slight edge. The results on the data sets with many missing values show where this small advantage is gained. Although it is not possible to compare the average accuracy over the sixteen data sets with that of the seven in Table 5.2, the relative difference in the accuracy values for these seven is noteworthy. The average value method is comfortably more accurate on the data set with the most missing values—LA with 33.64% missing. On the remaining six data sets the difference between “average value” and “always fail” is between 0% and 1.1%, with only the VO and V1 data showing a lead for the latter. Although these differences are small HENRY appears to benefit from the “average value” approach. Further testing on data sets with differing

	MISSING	ALWAYS FAIL	AVERAGE VALUE
HE	5.39%	78.7	79.8
HO	22.77%	73.9	74.7
HY	6.48%	98.6	99.0
LA	33.64%	76.2	82.5
SE	6.48%	96.8	96.8
V1	5.47%	87.7	87.4
VO	5.30%	94.8	94.7
mean		86.7	87.8

Table 5.2 Accuracy results for two missing value evaluation methods on seven data sets with more than 5% missing values.

amounts of missing values is necessary to determine the true extent of this apparent advantage.

5.2 Experiments on relational data sets

This section discusses four experiments using relational data. The first two, the arches problem and the trains problem, are presented by Quinlan in his original discussion of FOIL (Quinlan, 1990). The results for FOIL are repeated, and compared with HENRY. The next problem involves classification of small excerpts from artificial report abstracts. The objective is to generate a concept description that classifies each abstract as being related or unrelated to machine learning, and is based on an experiment described by Cohen (1995). The objective of the final experiment is to specify a set membership constraint given background knowledge about a set of integers.

In all these experiments the entire data set is used for training, and ABE is not used. Although ABE is capable of evaluating relational ripple-down rules, the emphasis is on comparing the form of the concept descriptions generated by HENRY and FOIL. All the data sets are relatively small compared to the UCI data sets, and both algorithms classify most examples correctly. Because relational data sets are typically generated from the target concept that the algorithms are trying to recognise, only a few representative training examples are used in each case.

The arches problem

The arches problem was discussed by Winston in his work on structured learning and is often used as a example classification task (Gennari *et al*., 1990; Quinlan, 1990). The objective is to create a concept description that distinguishes objects consisting of three blocks that form a valid arch from objects that are not arches. Winston provided four such objects (Figure 3.20), two that were arches, and two "near misses" that differed only slightly from the valid arches. The task's domain contains the following relations:

arch(A, B, C)	blocks A, B, and C form an arch with lintel A
supports(A, B)	block A supports block B
left-of(A, B)	block A is to the left of block B
touches(A, B)	the sides of blocks A and B touch
brick(A)	block A is a brick
wedge(A)	block A is a wedge
parallelepiped(A)	block A is a brick or a wedge

A set of extensional definitions based on these relations provides background information about the twelve blocks that make up the four objects.

FOIL's solution

The data set used by FOIL is shown in the appendix (file arches.foil). This file displays the extensional definitions that provide the background information for the task. The first two lines provide type information for the relations defined later, with the twelve blocks are identified as type *block*. The definition for the *arch* relation shows the two objects that are positive examples of the concept. Because no negative examples are given, FOIL uses the closed-world assumption to determine that all other combinations of *block*s are negative examples. The two near misses in Figure 3.20 are unnecessary, and their presence in the data can in fact restrict FOIL by reducing the number of negative examples.

FOIL learns the following definition for *arch*:

arch(A,B,C) :- leftof(B,C), supports(B,A), not(touches(B,C)).

This description is intuitive, and sufficiently general to cover other examples of *arch*. The general-to-specific nature of the covering algorithm is illustrated in the absence of literals such as *supports(C,A)* or *parallelepiped(A),* which are unnecessary to distinguish the training examples.

HENRY's solution

HENRY requires three input files to specify this data set. The first defines the combinations of the twelve blocks that constitute valid blocks, and is shown in the appendix (file arches.arff). In this file each block is an attribute of the relation, and examples are described by these twelve attributes. The object comprising blocks a1, b1, and c1 is an arch, and is identified by the tuple

pos, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0

where a 1 indicates the presence of the corresponding block, and a 0 indicates its absence. The other arch is similarly defined. Because HENRY does not use the closed-world assumption, it is necessary to provide negative examples of the concept.

The background relations are provided in two files, one for unary and one for binary relations. The first "attribute" in each of these files lists the names of the relations defined in the files, and the remaining attributes specify the objects that may appear as their arguments. This is equivalent to the FOIL input file's type specification.

From this data HENRY learns the definition for *arch* shown in Figure 5.2. The first two relations are identical to those used by FOIL, indicating their importance to the concept. However, this definition differs from FOIL's in three ways. First, HENRY's concept description is a ripple-down rule tree, although its single internal node is computationally similar to the rule generated by FOIL. The small size is of the concept description is a theme that permeates these experiments.

The second difference is inside the relational terms in the rule. The first term specifies that an example can only match *left-of(A B)* if it has two attributes (blocks) with the value 1 that also appear in one of the extensional definitions of *left-of*. The example illustrated earlier matches this term because it has the value 1 for attributes B1 and C1, and the definition *left-of(B*1*, C*1*)* is present in the binary relation file. The second term is satisfied in a similar manner, although the block in the first argument of *left-of* must be the first argument for the definition of *supports*.

The final difference between the two concept descriptions occurs in the third term, $A3 = 0$. Because HENRY does not use negative literals, it is unable to find the term *not(touches(B,C))*. The term HENRY selects is a propositional term, and its *m*-value would be identical to many others. This term says that a collection of blocks is an arch if it does not contain block A3. (Remember that a 0 indicates a block is not present.) The actual value chosen for this term is arbitrary because the absence of any one of six blocks would give the same result.

This last term illustrates a deficiency in HENRY's search. By introducing the constant value A3, HENRY has produced a concept description that is less generally applicable than FOIL's. If HENRY had been able to search the space of negated literals it too may have been able to use the *touches* relation. Although this term would have the same *m*-value as the propositional term chosen here, HENRY's preference for relational terms would have seen it retained.

The trains problem

The trains problem is also described by Quinlan in his discussion of FOIL. In this task each learning algorithm is presented with descriptions of ten trains, and must generate a concept description distinguishing trains travelling east from those travelling west. Trains consist of an engine and a number of cars, and the following relations define features of these objects:

eastbound(T)	train T is eastbound
has-car(T,C)	C is a car of train T
infront(C,D)	car C is in front of car D

```
 IS (A = 1 AND B = 1 AND left-of(A B)) AND (A = 1 AND C = 1 AND supports(A C)) AND
A3 = 0 ?
 No     Yes -> class = pos
 |
 class = neg
```

Figure 5.2 Relational ripple-down rules generated by HENRY for the arches problem.

long(C)	car C is long
open-rectangle(C)	car C is shaped as an open rectangle
...	similar relations for five other shapes
jagged-top(C)	car C has a jagged top
sloping-top(C)	car C has a sloping top
open-top(C)	car C has an open top
contains-load(C,L)	car C contains load L
1-item(C)	car C has one load item
...	similar relations for two and three load items
2-wheels(C)	car C has two wheels
3-wheels(C)	car C has three wheels

A set of extensional definitions based on these relations provides background information about the thirty cars that make up the ten trains in the problem. This learning task illustrates the kind of objects with different structures that are difficult to represent in propositional form.

FOIL's solution

The input file for FOIL is shown in the appendix (file trains.foil), and is similar to that of the arches problem. However, a set of negative examples of the target concept *eastbound* are provided. These negative examples are trains that are westbound, and serve to reduce the size of the search space by enumerating every possible train.

From this data FOIL finds the definition:

eastbound(A) :- has-car(A,B), not(long(B)), not(open-top(B)).

This concept description is again sufficient to cover all the *eastbound* examples, but the more specific nature of this learning task makes it less generally applicable than the arches definition.

HENRY's solution

HENRY again requires three ARFF files to specify the data for this experiment (files trains.arff, train_1rel.arff, and train_2rel.arff). In this case the attributes in the first ARFF file are similar to the type *c* in the FOIL input file. Each train is defined by a tuple indicating its direction, eastward or westward, and the presence or absence of each of the thirty cars. The other two files specify the other unary and binary background relations. From this data HENRY generates the concept description in Figure 5.3.

Because *eastbound* appears first in the list of class values, and there are equal numbers of each class, it is used as the default class, and HENRY learns a rule to describe

westbound trains. FOIL6 learns the following definition for *westbound* (this is achieved by swapping the negative and positive examples in the *eastbound* definition):

westbound(A) :- has-car(A,B), jagged-top(B).

Quinlan (1990) gives a different definition, which may be the result of differences in the version of the FOIL implementations used. Quinlan's definition for *westbound* is:

```
IS (A = 1 AND  long (A)) AND c11 = 0 AND c31 = 0 AND c51 = 0 ?
No     Yes -> class = westbound
|
class = eastbound
```

Figure 5.3 Relational ripple-down rules generated by HENRY for *westbound* trains.

westbound(A) :- has-car(A,B), long(B), 2-wheels(B), not(open-top(B)).

The first difference between HENRY's definition and FOIL's is the absence of the *has-car* relation. This relation is not defined in HENRY's data set because the information is given along with the class value in the trains.arff file. The *has-car* relation provides a link between the variable representing the train in the head of the FOIL definition with the variable representing the car in the body of the clause. The definitions generated by HENRY assume all objects identified in the clause are cars. The second literal in Quinlan's *westbound* definition is the same as the first in HENRY's definition. Following that the problem with negative literals, discussed in the previous section, arises again.

Figure 5.4 shows the concept description generated by HENRY with the order of the class values reversed. This is HENRY's equivalent to FOIL's *eastbound* definition and quite obviously suffers from not having negated terms. Allowing HENRY to search through the space of negative terms would be beneficial for classification tasks such as this and the arches problem.

```
IS (A = 1 AND  three-wheels (A)) AND c81 = 0 ?
No     Yes -> class = eastbound
|
IS c21 = 1 ?
No     Yes -> class = eastbound
|
IS c41 = 1 ?
No     Yes -> class = eastbound
|
class = westbound
```

Figure 5.4 Relational ripple-down rules generated by HENRY for *eastbound* trains.

The text classification problem

```
IS (A = "machine" AND B = "learning" AND  "succ" (A B)) ?
No     Yes
 |      |
 |      IS "0" = "like" ?
 |      No     Yes -> "class" = "neg"
 |      |
 |      "class" = "pos"
 |
IS (C = "induce" AND D = "decision" AND  "succ" (C D)) AND
   (E = "decision" AND F = "trees" AND  "succ" (E F)) ?
No     Yes -> "class" = "pos"
 |
"class" = "neg"
```

Figure 5.5 Relational ripple-down rules generated by HENRY for the text classification problem.

The text classification problem is similar to that introduced in Section 3.1. The data represents the first ten words of thirteen artificial technical report abstracts, five of which are about machine learning (the target concept). The abstracts are intended to be difficult for a propositional scheme to distinguish using data that only indicates the appearance of words. The planned definition specifies positional relationships between important words in the text.

The learning algorithms are given the following background relations defining positional relationships between words:

ml_document(D)	document D is about machine learning
succ(A,B)	word B immediately follows word A

The words themselves are identified by the document they occur in and their position in that document:

c4.5(D,N)	“c4.5” is the Nth word of document D
ability(D,N)	“ability” is the Nth word of document D
...	similar definitions for every other word in the text

A set of extensional definitions based on these relations provides background information about the positions of the words in the thirteen abstracts.

FOIL’s solution

The FOIL input file contains the definitions as they are described above. Each word becomes a relation name, with an extensional definition for every occurrence of the word in an abstract. From this data FOIL learns the following definition for the *ml_document*:

ml_document(A) :- algorithm(A,B), not(the(A,B)).

ml_document(A) :- as(A,B).

HENRY’s solution

HENRY requires two ARFF files for this problem. The first specifies the thirteen example abstracts as tuples indicating their class value and the word that appears in each position in the abstract. The attributes pertaining to word positions are declared with every word from the text because HENRY only compares symbolic attributes declared with the same set of symbols. The second data file contains extensional definitions for the *successor* relation over the first ten word positions. Figure 5.5 shows the concept description generated by HENRY.

In this task HENRY has learned a concept description that appears more intuitive than FOIL's. Although FOIL's solution is simpler, it is more specific to the training examples than HENRY's RDR tree. HENRY has recognised two important phrases that appear in the machine learning abstracts. The first, "machine learning", accounts for three positive examples and one negative example, hence the exception to the first rule that specifically identifies the negative example as having "like" as its first word (position 0). The second phrase is "induce decision trees", which covers the other two positive examples.

HENRY's version of the text data differs in form from the data sets presented in the previous two experiments, where the values in the propositional data set indicated the presence of absence of an object, and were therefore equivalent to the truth values *true* and *false*. In this data set the values can be any of the words that appear in the text. This removes the need to define the type *document* that provides the link between the target relation and the background relations in the purely relational form. It would be possible to represent this data set in a manner similar to that used for FOIL, with the propositional data set mapping each document's class value to an object of type *document*. The first rule in the RDR tree might then look like:

IS A = "t" ... AND machine(A B) ... AND learning(A C) ... AND succ(B C) ?

Here the variable *A* refers to an object of type document and the variables *B* and *C* refer to objects of type word. The next task shows a data set expressed using the "mapping" approach.

The numbers problem

The objective of this task is to identify a set membership constraint for a group of integers, ranging from 0 to 18. Background information is in the form of four mathematical properties, which each learning algorithm must use to determine the membership criterion for a given subset of the integers. The following relations are used to specify these properties:

prime(A) integer A is a prime

odd(A) integer A is odd

```
IS (A = "t" AND  "prime" (A)) AND (A = "t" AND  "less_than" (A "12"))?
No     Yes -> "class" = "pos"
|
"class" = "neg"
```

Figure 5.6 Relational ripple-down rules generated by HENRY for the numbers problem.

even(A)	integer A is even
less_than(A,B)	integer A is less than integer B

A set of extensional definitions using these relations defines the properties of the nineteen integers in the expected manner.

FOIL's solution

The FOIL input file is shown in the appendix (file nums3.foil). The name of the target definition (indicated by the absence of an asterisk in front of its declaration) betrays the intended membership criterion. The type *num* defines the nineteen integers, and these appear as positive and negative examples of the target relation as expected. The remainder of the file defines the other mathematical properties.

From this data FOIL generates the following definition for *primeslessthan12*:

primelessthan12(A) :- prime(A), lessthan(A,B), lessthan(B,C), lessthan(C,D), prime(B).

This definition is true for all five positive training examples, but it is intuitively bad. The addition of new negative examples (and positive examples if any existed) would cause it to fail. Even though it uses no constant values, the definition is specific to the number of training examples.

HENRY's solution

HENRY again requires three arff files for this data. The first defines the nineteen integers (file nums3.arff), and the others (nums3_1rel.arff and nums3_2rel.arff) define the unary and binary background relations in the usual way. The file defining the integers does no more than map each integer value to an attribute name, and specify whether or not the integer is a member of the target concept. It is a grid with a diagonal line of "*t*"s showing where the *n*th column intersects the *n*th row. From this data HENRY generates the concept description in Figure 5.5.

An integer *N* matches this definition if it has a *"t"* for an attribute that has definitions for *prime* and *less_than(N,"12")*—HENRY has found the intended set membership constraint of "prime and less than 12". Although the rule contains a constant value, the definition would endure with new negative examples. This task

indicates how HENRY can combine variables and constant values in relational terms, and how this can be beneficial in some classification tasks.

6.

Conclusions

HENRY shows that both a probabilistic search heuristic and a tree-like concept description format can be used successfully in relational learning. Despite the experimental nature of the implementation it is capable of managing, in a reasonable amount of time, a number of the standard relational learning tasks presented in the literature. However, some characteristics of the ripple-down rule structure place other tasks, learnable by inductive logic programs, outside HENRY's scope. Some limitations in the implementation, imposed largely by the project's timespan, also make some of the concept descriptions generated by the program less than satisfactory.

Combining propositional and relational terms

One of the main advantages HENRY has over standard inductive logic programming systems is its ability to combine both propositional and relational terms in a single rule. Relational terms using a mixture of variables and constant values are also outside the restrictions imposed by the ILP framework, yet their use by HENRY is shown to be beneficial in one experiment. The use of different-order terms is not limited to ripple-down rules, and HENRY can also use them in DNF concept descriptions. However, ripple-down rules provide a more concise description, as shown with the iris data set.

The ability to combine relational and propositional terms means that HENRY can apply three relational operators to purely propositional data. These three relations, equality for symbolic attributes and two inequalities for numeric data, can be used on any pair of attributes of the same type, and increase the classification accuracy of the concept description for the iris data by 4%. HENRY's performance on the other UCI data sets using these relations is less remarkable, and may be due to the character of the individual attributes. In the iris data set all the attributes (apart form the class) are numeric, and all are measurements of length using the same unit. Therefore, these attributes are suitable for comparison using the < and relations, as evidenced by the increase in accuracy. The other data sets, however, often have a combination of numeric and symbolic attributes, the units of which are incompatible for comparison. For example, there is little point in directly comparing an attribute representing time with one representing weight. Most symbolic attributes are also incompatible for direct comparison even when they share the same set of symbols. The assignment of the symbols is arbitrary and a different assignment would result in a different set of relational regularities appearing in the data.

At present HENRY compares every pair of numeric attributes and every pair of symbolic attributes in a data set if the user specifies the use of the three standard relations. It would be better to assume that no pair of attributes are comparable, and

require the user to specify sets of attributes that share the same meaning. This would reduce the search space of the algorithm, and also eliminate the possibility of relationships that occur by chance being discovered.

Experimental results

The relational experiments presented in Chapter 5 show that HENRY is able to learn concept descriptions similar to those learned by FOIL, given a similar set of background relations. In most cases the initial positive literals in FOIL's definition of a concept are mirrored in HENRY's ripple-down rules. However, the nature of the learning tasks highlights deficiencies in HENRY's current implementation. The tasks were selected to illustrate the capabilities of FOIL, and show the range of problems it is able to solve. Part of this range involves learning recursive definitions for such concepts as list membership and reversal. HENRY is unable to learn recursive definitions using ripple-down rules, because the structure cannot make reference to itself. Moreover, the feasibility of multiple-tree concept descriptions is equivocal. Structures of this form might be necessary to attain the same level of meaning expressed in a multi-clause logic program, but it is unclear how such concept definitions would be generated and evaluated.

Negated literals

Another feature that FOIL finds advantageous is the use of negated literals, which allows the algorithm to practically double the number of relations it can search through. Empowering HENRY with such an ability is possible, and it would have additional benefits. Presently the program uses three propositional operators, <, , and =. A negation operator would allow the set of operators to be replaced with <, *not* < (same as), =, and *not* =. This would provide a certain symmetry between the symbolic and numeric operators, although the worth of "*not equal*" is not obvious.

Another approach to this issue would be to re-express many of the learning problems that make use of negated literals so that the reciprocal of the relation is stated in the background knowledge. In the arches problem, the *touches* relation could be replaced with an extensional definition for *not_touches* or *gap_between*. This relation would then be used in the clauses defining *arch* producing what might be considered a more satisfactory solution. The important feature of the upright blocks in an arch is that there is a gap between them, and this should be expressed in a positive manner. It is possible to think of many relationships in addition to *touches* that are *not* true for a pair of blocks.

The other side of this issue is concerned with reducing the number of extensional definitions required to specify a relation such as *touches*. With the intent of using the negated form of this relation, FOIL need only be given knowledge about the few blocks that do touch. The extensional definition for *gap_between* would contain every

combination of blocks not covered by *touches*. In some situations re-expressing a relation in this manner might be impractical, and could also interfere with the use of the closed-world assumption.

Ripple-down rules

The search for relational terms increases the search space for any learning task. FOIL employs several restrictions to limit its search, but the space explored is still large. Partly for this reason, and partly because they are designed around a simple predefined target relation, the example relational learning tasks presented in the literature tend to be very small. The concept descriptions defining these relations often contain one or two clauses, with only a few literals each. The simplicity of the target relations makes redundant many of the features of ripple-down rules. HENRY is able to express the target concept in many cases as an RDR tree with a single rule. The *if_false* branch of the tree classifies an example as the default class, and the *if_true* branch is also empty.

The ripple-down rule structure gains much of its advantage over traditional rule formats in propositional domains. Here, data sets often contain many thousands of examples described by tens or hundreds of attributes, and the diffuse nature of the target concepts often necessitates large rule sets or decision trees for accurate representation. Ripple-down rules excel in these situations by removing much of the redundancy present in DNF rules. The binary nature of the logic used to match examples to rules means that any one rule can only distinguish examples of a single class. This is not a problem for an algorithm learning ripple-down rules because it defers discrimination of all examples not matching the current rule to the *if_false* subtree of the rule. A DNF learner in the same situation may have to duplicate large parts of rules to cover each class in turn.

The definition of a covering algorithm means it must generate a description for every class in the target concept. If there are three classes, as in the iris data, the algorithm must generate at least one definition for each class. However, the definition of inductive logic programming means that ILP systems need only learn concepts that have two classes. These concepts define a set of objects that belong to the target relation, so the two classes for any problem are the set of objects belonging to the target relation, and the set not belonging to the relation. The computational meaning of logic programming implies an ILP system need only present a definition for membership to the target relation. Any object not matching this definition is not a member of the target relation by the negation-as-failure rule.

The use of two-class domains for relational learning is another factor in the limited application of ripple-down rules to these tasks. Ripple-down rules that describe a two-class concept can often degenerate into a list-like binary tree, where examples are continually

passed down *if_false* branches, and classified by the first rule they match. The RDR tree in Figure 4.7 shows how a multi-class problem can produce a more balanced tree. To learn an equivalent concept two-class algorithms such as FOIL must generate three definitions, one for each cultivar of iris in turn. Classification of new examples then requires matching against all three concept descriptions, the combination of which may have a number of redundant clauses.

The ripple-down rule structure removes two of the uncertainties that arise when evaluating DNF rules. Examples that match two or more rules in a DNF rule set may be given more than one classification, and examples that match none of the rules will be given no classification. Ripple-down rules avoid these problems by providing default classifications for examples that do not match any rules in the tree. The tree-like structure also ensures that examples can follow only a single path to a leaf node and be given a single classification. Inductive logic programming systems overcome these problems by producing rules that define a single class, thus eliminating the possibility of multiple classifications. Any examples not matching any of the clauses in the definition are simply classified as negative examples of the target concept, and no examples are left uncategorised.

Missing values

Missing values can be a cause of new examples not matching rules and being misclassified. The traditional approach to this situation is to assume the missing value would not have matched the term in question, and the example fails to match the rule. If an example is missing a value for every attribute, an RDR concept description will see it classified as the most frequent class in the training data—the example will fail to match every rule and will be given the default definition of the top-most rule. This method assigns an "unclassifiable" example to the class that is most likely to be correct, given no information about the example. An alternative approach is to assume the example is similar to the other examples arriving at term that evokes the missing value. This method assumes the example has the average of the values seen at a given point in the RDRs, and uses that value instead of simply failing to match the term. The "average example" approach allows examples that are missing only a few values to match rules that also match examples similar in terms of other attributes. Experimental results show that this method is at least as good as the "always fail" technique, and improves the performance of the concept description on data sets with many missing values.

In combination with the other features of HENRY, this approach to missing values establishes the algorithm as an auspicious addition to the field of machine learning. With additional work on the missing value issue, and rectification of the other shortcomings

identified in the implementation, HENRY should prove successful at combining the duties relational and propositional learning.

References

Aha, D.W., Kibler, D. and Albert, M.K. (1991) "Instance-Based Learning Algorithms", *Machine Learning* 6, pp. 37–66.

Bratko, I. (1990) *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham; 2nd edition.

Breiman, L., Friedman, J.H., Olshen and R.A., Stone, C.J. (1984) *Classification and Regression Trees*. Wadsworth Inc., Belmont, California.

Cameron-Jones, R.M. and Quinlan, J.R. (1993) "Avoiding Pitfalls When Learning Recursive Theories", *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp.1050–1055.

Cendrowska, J. (1987) "PRISM: An Algorithm for Inducing Modular Rules", *International Journal of Man–Machine Studies* 27(4), pp. 349–370.

Cleary, J.G. and Trigg, L.E. (1995) "K*: An Instance-based Learner Using an Entropic Distance Measure", *Proceedings of the 12th International Conference on Machine Learning*, Tahoe City, California.

Cohen, P.R. and Feigenbaum, E.A., editors (1982) "Version Space", in *The handbook of artificial intelligence, Vol. III*, pp.385–400.

Cohen, W.W. (1995) "Text Categorization and Relational Learning", *Proceedings of the 12th International Conference on Machine Learning*, Tahoe City, California.

Compton, P. and Jansen, R. (1990) "Knowledge in context: A strategy for expert system maintenance", *Proceedings of AI'88: 2nd Australian Joint Artificial Intelligence Conference*, pp. 292–306.

Copi, I.M. (1979) *Symbolic Logic*. Macmillan, New York; 5th edition.

Cunningham, S. and Summers, B. (1995) "Applying machine learning to subject classification and subject description for information retrieval", Working Paper 95/20, Department of Computer Science, The University of Waikato, Hamilton, New Zealand.

Dzeroski, S. and Lavrac, N. (1992) "Refinement Graphs for FOIL and LINUS", in *Inductive Logic Programming*, S. Muggleton ed., Academic Press, London.

Fürnkranz, J. (1994) "Efficient Pruning Methods for Relational Learning", Technical Report OEFAI-TR-94-28, Austrian Research Institute for Artificial Intelligence, Vienna.

Gaines, B.R. (1991) "The Tradeoff Between Knowledge and Data in Knowledge Acquisition." In *Knowledge Discovery in Databases*, G. Piatetsky-Shapiro and W.J. Frawley, eds. AAAI Press, Menlo, California, pp. 491–505.

Gaines, B.R. and Compton, P. (1995) "Induction of Ripple-Down Rules Applied to Modeling Large Databases", *Journal of Intelligent Information Systems* 4.

Garner, S.R., Cunningham, S., Holmes, G., Nevill-Manning, C., Witten, I.H. (1995) "Machine Learning in Practice: Experience with Agricultural Databases", Machine Learning in Practice Workshop, at the 12th International Conference on Machine Learning, Tahoe City, California.

Holmes, G., Donkin, A. and Witten, I.H. (1994) "WEKA: A Machine Learning Workbench", Department of Computer Science, The University of Waikato, Hamilton, New Zealand.

Holte, R. (1993) “Very simple classification rules perform well on most commonly used datasets”, *Machine Learning* 11(1), pp. 63–90.

Langley, P. and Simon, H.A. (1995) “Applications of Machine Learning and Rule Induction”, To appear in *Communications of the ACM*.

Lavrac, N. and Dzeroski, S. (1994) *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood Ltd, Hemel Hempstead, Hertfordshire.

Ling, C.X. (1992) “Logic Program Synthesis from Good Examples”, in *Inductive Logic Programming*, S. Muggleton ed., Academic Press, London.

Lloyd, J.W. (1987) *Fundamentals of Logic Programming*. Springer-Verlag, Berlin; 2nd edition.

Mooney, R.J. (1992) “Encouraging Experimental Results on Learning CNF”, Technical Report, University of Texas.

Muggleton, S. (1991) “Inductive Logic Programming”, *New Generation Computing* 8 (4), pp.295–318.

Muggleton, S., editor (1992) *Inductive Logic Programming*. Academic Press, London.

Muggleton, S. and Feng, C. (1992) “Efficient induction of logic programs”, in *Inductive Logic Programming*, S. Muggleton ed., Academic Press, London.

Pazzani, M.J., Brunk, C.A. and Silverstein, G. (1992) “An Information-Based Approach to Integrating Empirical and Explanation-Based Learning”, in *Inductive Logic Programming*, S. Muggleton ed., Academic Press, London.

Pazzani, M.J. and Kibler, D. (1992) “The Utility of Knowledge in Inductive Learning”, *Machine Learning* 9, pp. 57–94.

Quinlan, J.R. (1986) “Induction of Decision Trees”, *Machine Learning* 1, pp.81–106.

Quinlan, J.R. (1990) “Learning Logical Definitions from Relations”, *Machine Learning* 5, pp.239–266.

Quinlan, J.R. (1993) *C4.5: Programs for Machine Learning*. Morgan Kaufman.

Quinlan, J.R. (1995) “MDL and Categorical Theories (Continued)”, *Proceedings of the 12th International Conference on Machine Learning*, Tahoe City, California.

Quinlan, J.R. and Cameron-Jones, R.M. (1994) “Living in a Closed World”, Basser Department of Computer Science, The University of Sydney.

Salzberg, S.L. (1990) *Learning With Nested Generalized Exemplars*. Kluwer Academic Publishers, Norwell, Massachusetts.

Salzberg, S.L. (1995) “On Comparing Classifiers: A Critique of Current Research and Methods”, Technical Report JHU-95/06, Department of Computer Science, Johns Hopkins University.

Utgoff, P.E. (1989) “Incremental induction of decision trees”, *Machine Learning* 4, pp. 161–186.

Appendix

The UCI datasets

Dataset	Examples	Attributes	Missing values	Class values	Notes
BC	286	10	9	2	Recurrence of breast cancer
CH	3196	37	0	2	Chess end games
GL	214	10	0	7	Glass types, no examples of class 4
G2	163	10	7	2	GL with classes 1 and 3 combined, classes 4-7 removed
HD	303	14	7	5	Heart disease
HE	155	20	167	2	Hepatitis
HO	368	23	1927	2	Horse cholic
HY	3163	26	5329	2	Thyroid disease - hypothyroid
IR	150	5	0	3	Iris
LA	57	17	326	2	Labour negotiations
LY	148	19	0	4	Lymphography
MU	8124	23	2480	2	Mushrooms
SE	3163	26	5329	2	Thyroid disease - sick-euthyroid
SO	47	36	0	4	Soybean diseases
VO	435	16	392	2	Voting records
V1	435	17	381	2	VO with attribute “physician-fee-freeze” removed

The *arches* dataset

arches.arff

```
@relation arches
@attribute class {neg,pos}
@attribute A1 {1,0}
@attribute A2 {1,0}
@attribute A3 {1,0}
@attribute A4 {1,0}
@attribute B1 {1,0}
@attribute B2 {1,0}
@attribute B3 {1,0}
@attribute B4 {1,0}
@attribute C1 {1,0}
@attribute C2 {1,0}
@attribute C3 {1,0}
@attribute C4 {1,0}
@data
pos,1,0,0,0,1,0,0,0,1,0,0,0
pos,0,0,0,1,0,0,0,1,0,0,0,1

neg,1,0,0,0,1,0,0,0,0,1,0,0
neg,1,0,0,0,1,0,0,0,0,0,1,0
neg,1,0,0,0,1,0,0,0,0,0,0,1
neg,1,0,0,0,0,1,0,0,1,0,0,0
neg,1,0,0,0,0,1,0,0,0,1,0,0
neg,1,0,0,0,0,1,0,0,0,0,1,0
neg,1,0,0,0,0,1,0,0,0,0,0,1
neg,1,0,0,0,0,0,1,0,1,0,0,0
neg,1,0,0,0,0,0,1,0,0,1,0,0
neg,1,0,0,0,0,0,1,0,0,0,1,0
neg,1,0,0,0,0,0,1,0,0,0,0,1
neg,1,0,0,0,0,0,0,1,1,0,0,0
neg,1,0,0,0,0,0,0,1,0,1,0,0
neg,1,0,0,0,0,0,0,1,0,0,1,0
neg,1,0,0,0,0,0,0,1,0,0,0,1
neg,0,1,0,0,1,0,0,0,1,0,0,0
neg,0,1,0,0,1,0,0,0,0,1,0,0
neg,0,1,0,0,1,0,0,0,0,0,1,0
neg,0,1,0,0,1,0,0,0,0,0,0,1
neg,0,1,0,0,0,1,0,0,1,0,0,0
neg,0,1,0,0,0,1,0,0,0,1,0,0
neg,0,1,0,0,0,1,0,0,0,0,1,0
neg,0,1,0,0,0,1,0,0,0,0,0,1
neg,0,1,0,0,0,0,1,0,1,0,0,0
neg,0,1,0,0,0,0,1,0,0,1,0,0
neg,0,1,0,0,0,0,1,0,0,0,1,0
```

```
neg,0,1,0,0,0,0,1,0,0,0,0,1
neg,0,1,0,0,0,0,0,1,1,0,0,0
neg,0,1,0,0,0,0,0,1,0,1,0,0
neg,0,1,0,0,0,0,0,1,0,0,1,0
neg,0,1,0,0,0,0,0,1,0,0,0,1
neg,0,0,1,0,1,0,0,0,1,0,0,0
neg,0,0,1,0,1,0,0,0,0,1,0,0
neg,0,0,1,0,1,0,0,0,0,0,1,0
neg,0,0,1,0,1,0,0,0,0,0,0,1
neg,0,0,1,0,0,1,0,0,1,0,0,0
neg,0,0,1,0,0,1,0,0,0,1,0,0
neg,0,0,1,0,0,1,0,0,0,0,1,0
neg,0,0,1,0,0,1,0,0,0,0,0,1
neg,0,0,1,0,0,0,1,0,1,0,0,0
neg,0,0,1,0,0,0,1,0,0,1,0,0
neg,0,0,1,0,0,0,1,0,0,0,1,0
neg,0,0,1,0,0,0,1,0,0,0,0,1
neg,0,0,1,0,0,0,0,1,1,0,0,0
neg,0,0,1,0,0,0,0,1,0,1,0,0
neg,0,0,1,0,0,0,0,1,0,0,1,0
neg,0,0,1,0,0,0,0,1,0,0,0,1
neg,0,0,0,1,1,0,0,0,1,0,0,0
neg,0,0,0,1,1,0,0,0,0,1,0,0
neg,0,0,0,1,1,0,0,0,0,0,1,0
neg,0,0,0,1,1,0,0,0,0,0,0,1
neg,0,0,0,1,0,1,0,0,1,0,0,0
neg,0,0,0,1,0,1,0,0,0,1,0,0
neg,0,0,0,1,0,1,0,0,0,0,1,0
neg,0,0,0,1,0,1,0,0,0,0,0,1
neg,0,0,0,1,0,0,1,0,1,0,0,0
neg,0,0,0,1,0,0,1,0,0,1,0,0
neg,0,0,0,1,0,0,1,0,0,0,1,0
neg,0,0,0,1,0,0,1,0,0,0,0,1
neg,0,0,0,1,0,0,0,1,1,0,0,0
neg,0,0,0,1,0,0,0,1,0,1,0,0
neg,0,0,0,1,0,0,0,1,0,0,1,0
```

arch_1rels.arff

```
@relation 1-ary_rels
@attribute rel-def {brick,wedge,p-
ped}
@attribute a
{A1,A2,A3,A4,B1,B2,B3,B4,C1,C2,C3,C4}
@data
brick,A1
brick,A2
brick,A3
brick,B1
brick,B2
brick,B3
brick,B4
brick,C1
brick,C2
brick,C3
brick,C4
wedge,A4
p-ped,A1
p-ped,A2
p-ped,A3
p-ped,B1
p-ped,B2
p-ped,B3
p-ped,B4
p-ped,C1
p-ped,C2
p-ped,C3
p-ped,C4
p-ped,A4
```

arch_2rels.arff

```
@relation 2-ary_rels
@attribute rel-def {supports,left-
of,touches}
@attribute a
{A1,A2,A3,A4,B1,B2,B3,B4,C1,C2,C3,C4}
@attribute b
{A1,A2,A3,A4,B1,B2,B3,B4,C1,C2,C3,C4}
@data
left-of,B1,C1
left-of,B2,C2
left-of,B3,C3
left-of,B4,C4
supports,B1,A1
supports,C1,A1
supports,B3,A3
supports,C3,A3
supports,B4,A4
supports,C4,A4
touches,A1,B1
touches,B1,A1
touches,A1,C1
touches,C1,A1
touches,A2,B2
touches,B2,A2
touches,A2,C2
touches,C2,A2
touches,A3,B3
touches,B3,A3
touches,B3,C3
touches,C3,B3
touches,C3,A3
touches,A3,C3
touches,A4,B4
touches,B4,A4
touches,A4,C4
touches,C4,A4
```

arches.foil

```
#block:a1,b1,c1,a2,b2,c2,a3,b3,c3,a4
,b4,c4.

arch(block,block,block)
a1,b1,c1
a4,b4,c4
.
*leftof(block,block)
b1,c1
b2,c2
b3,c3
b4,c4
.
*supports(block,block)
b1,a1
c1,a1
b3,a3
c3,a3
b4,a4
c4,a4
.
*touches(block,block)
a1,b1
b1,a1
a1,c1
c1,a1
a2,b2
b2,a2
a2,c2
c2,a2
a3,b3
b3,a3
a3,c3
c3,a3
b3,c3
c3,b3
a4,b4
b4,a4
a4,c4
c4,a4
.
*brick(block)
a1
b1
c1
a2
b2
c2
a3
b3
c3
b4
c4
.
*wedge(block)
a4
.
*parallelepiped(block)
a1
b1
c1
a2
b2
c2
a3
b3
c3
a4
b4
c4
.
```

The *trains* dataset

trains.arff

```
@relation trains
@attribute class {eastbound,westbound}
@attribute c11 {1,0}
@attribute c12 {1,0}
@attribute c13 {1,0}
@attribute c14 {1,0}
@attribute c21 {1,0}
@attribute c22 {1,0}
@attribute c23 {1,0}
@attribute c31 {1,0}
@attribute c32 {1,0}
@attribute c33 {1,0}
@attribute c41 {1,0}
@attribute c42 {1,0}
@attribute c43 {1,0}
@attribute c44 {1,0}
@attribute c51 {1,0}
@attribute c52 {1,0}
@attribute c53 {1,0}
@attribute c61 {1,0}
@attribute c62 {1,0}
@attribute c71 {1,0}
@attribute c72 {1,0}
@attribute c73 {1,0}
```

```
@attribute c81 {1,0}
@attribute c82 {1,0}
@attribute c91 {1,0}
@attribute c92 {1,0}
@attribute c93 {1,0}
@attribute c94 {1,0}
@attribute c101 {1,0}
@attribute c102 {1,0}
@data
eastbound,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
eastbound,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
eastbound,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
eastbound,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
eastbound,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0
westbound,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0
westbound,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0
westbound,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0
westbound,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0
westbound,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1
```

train_1rel.arff

```
@relation 1-ary
@attribute rel {long,open-
top,jagged-top,sloping-top,one-
item,two-items,three-items,two-
wheels,three-wheels}
@attribute a1
{c11,c12,c13,c14,c21,c22,c23,c31,c32,
c33,c41,c42,c43,c44,c51,c52,c53,c61,c
62,c71,c72,c73,c81,c82,c91,c92,c93,c9
4,c101,c102}
@data
long,c11
long,c13
long,c33
long,c52
long,c61
long,c73
long,c81
long,c92
long,c102
open-top,c11
open-top,c13
open-top,c14
open-top,c21
open-top,c22
open-top,c31
open-top,c41
open-top,c42
open-top,c44
open-top,c51
open-top,c62
open-top,c71
open-top,c72
open-top,c82
open-top,c91
open-top,c93
open-top,c94
open-top,c101
open-top,c102
jagged-top,c73
jagged-top,c92
sloping-top,c12
one-item,c12
one-item,c13
one-item,c14
one-item,c21
one-item,c22
one-item,c31
one-item,c32
one-item,c33
one-item,c41
one-item,c42
one-item,c43
one-item,c44
one-item,c51
one-item,c52
one-item,c53
one-item,c62
one-item,c71
one-item,c72
one-item,c81
one-item,c82
one-item,c91
one-item,c92
one-item,c93
one-item,c94
one-item,c101
two-items,c23
two-items,c102
three-items,c11
three-items,c61
two-wheels,c11
two-wheels,c12
two-wheels,c14
two-wheels,c21
two-wheels,c22
two-wheels,c23
two-wheels,c31
two-wheels,c32
two-wheels,c41
two-wheels,c42
two-wheels,c43
two-wheels,c44
two-wheels,c51
two-wheels,c53
two-wheels,c61
two-wheels,c62
two-wheels,c71
two-wheels,c72
two-wheels,c73
```

```
two-wheels,c82
two-wheels,c91
two-wheels,c92
two-wheels,c93
two-wheels,c94
two-wheels,c101
two-wheels,c102
three-wheels,c13
three-wheels,c33
three-wheels,c52
three-wheels,c81
```

trains_2rel.arff

```
@relation 2-ary
@attribute rel {infront}
@attribute a
{c11,c12,c13,c14,c21,c22,c23,c31,c32,
c33,c41,c42,c43,c44,c51,c52,c53,c61,c
62,c71,c72,c73,c81,c82,c91,c92,c93,c9
4,c101,c102}
@attribute b
{c11,c12,c13,c14,c21,c22,c23,c31,c32,
c33,c41,c42,c43,c44,c51,c52,c53,c61,c
62,c71,c72,c73,c81,c82,c91,c92,c93,c9
4,c101,c102}
@data
infront,c11,c12
infront,c12,c13
infront,c13,c14
infront,c21,c22
infront,c22,c23
infront,c31,c32
infront,c32,c33
infront,c41,c42
infront,c42,c43
infront,c43,c44
infront,c51,c52
infront,c52,c53
infront,c61,c62
infront,c71,c72
infront,c72,c73
infront,c81,c82
infront,c91,c92
infront,c92,c93
infront,c93,c94
infront,c101,c102
```

trains.foil

```
#t:t1,t2,t3,t4,t5,t6,t
7,t8,t9,t10.
#c:c11,c12,c13,c14,c21
,c22,c23,c31,c32,c33,c
41,c42,c43,c44,c51,c52
,c53,c61,c62,c71,c72,c
73,c81,c82,c91,c92,c93
,c94,c101,c102.

eastbound(t)
t1
t2
t3
t4
t5
;
t6
t7
t8
t9
t10
.
*has-car(t,c)
t1,c11
t1,c12
t1,c13
t1,c14
t2,c21
t2,c22
t2,c23
t3,c31
t3,c32
t3,c33
t4,c41
t4,c42
t4,c43
t4,c44
t5,c51
t5,c52
t5,c53
t6,c61
t6,c62
t7,c71
t7,c72
t7,c73
t8,c81
t8,c82
t9,c91
t9,c92
t9,c93
t9,c94
t10,c101
t10,c102
.
*infront(c,c)
c11,c12
c12,c13
c13,c14
c21,c22
c22,c23
c31,c32
c32,c33
c41,c42
c42,c43
c43,c44
c51,c52
c52,c53
c61,c62
c71,c72
c72,c73
c81,c82
c91,c92
c92,c93
c93,c94
c101,c102
```

```
.
*long(c)
c11
c13
c33
c52
c61
c73
c81
c92
c102
.
*open-top(c)
c11
c13
c14
c21
c22
c31
c41
c42
c44
c51
c62
c71
c72
c82
c91
c93
c94
c101
c102
.
*jagged-top(c)
c73
c92
.
*sloping-top(c)
c12
.
*one-item(c)
c12
c13
c14
c21
c22
c31
c32
c33
c41
c42
c43
c44
c51
c52
c53
c62
c71
c72
c81
c82
c91
c92
c93
c94
c101
.
*two-items(c)
c23
c102
.
*three-items(c)
c11
c61
.
*two-wheels(c)
c11
c12
c14
c21
c22
c23
c31
c32
c41
c42
c43
c44
c51
c53
c61
c62
c71
c72
c73
c82
c91
c92
c93
c94
c101
c102
.
*three-wheels(c)
c13
c33
c52
c81
.
```

The *numbers* data

nums3.foil

```
#num:0,1,2,3,4,5,6,7,8
,9,10,11,12,13,14,15,1
6,17,18.

primelessthan12(num)
2
3
5
7
11
;
0
1
4
6
8
9
10
12
13
14
```

```
15
16
17
18
.
*even(num)
0
2
4
6
8
10
12
14
16
18
.
*odd(num)
1
3
5
7
9
11
13
15
17
.
*prime(num)
2
3
5
7
11
13
17
.
*lessthan(num,num)
0,1
0,2
0,3
0,4
0,5
0,6
0,7
0,8
0,9
0,10
0,11
0,12
1,2
1,3
1,4
1,5
1,6
1,7
1,8
1,9
1,10
1,11
1,12
2,3
2,4
2,5
2,6
2,7
2,8
2,9
2,10
2,11
2,12
3,4
3,5
3,6
3,7
3,8
3,9
3,10
3,11
3,12
4,5
4,6
4,7
4,8
4,9
4,10
4,11
4,12
5,6
5,7
5,8
5,9
5,10
5,11
5,12
6,7
6,8
6,9
6,10
6,11
6,12
7,8
7,9
7,10
7,11
7,12
8,9
8,10
8,11
8,12
9,10
9,11
9,12
10,11
10,12
11,12
0,13
1,13
2,13
3,13
4,13
5,13
6,13
7,13
8,13
9,13
10,13
11,13
12,13
0,14
1,14
2,14
3,14
4,14
5,14
6,14
7,14
8,14
9,14
10,14
11,14
12,14
13,14
0,15
1,15
2,15
3,15
```

```
4,15
5,15
6,15
7,15
8,15
9,15
10,15
11,15
12,15
13,15
14,15
0,16
1,16
2,16
3,16
4,16
5,16
6,16
7,16
8,16
9,16
10,16
11,16
12,16
13,16
14,16
15,16
0,17
1,17
2,17
3,17
4,17
5,17
6,17
7,17
8,17
9,17
10,17
11,17
12,17
13,17
14,17
15,17
16,17
0,18
1,18
2,18
3,18
4,18
5,18
6,18
7,18
8,18
9,18
10,18
11,18
12,18
13,18
14,18
15,18
16,18
17,18
.
```

nums3.arff

```
@relation primes_less_than_12
@attribute "class" {"neg","pos"}
@attribute "0" {"t","f"}
@attribute "1" {"t","f"}
@attribute "2" {"t","f"}
@attribute "3" {"t","f"}
@attribute "4" {"t","f"}
@attribute "5" {"t","f"}
@attribute "6" {"t","f"}
@attribute "7" {"t","f"}
@attribute "8" {"t","f"}
@attribute "9" {"t","f"}
@attribute "10" {"t","f"}
@attribute "11" {"t","f"}
@attribute "12" {"t","f"}
@attribute "13" {"t","f"}
@attribute "14" {"t","f"}
@attribute "15" {"t","f"}
@attribute "16" {"t","f"}
@attribute "17" {"t","f"}
@attribute "18" {"t","f"}
@data
"neg","t","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f"
"neg","f","t","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f"
"pos","f","f","t","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f"
"pos","f","f","f","t","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f"
"neg","f","f","f","f","t","f","f","f","f","f","f","f","f","f","f","f","f","f","f"
"pos","f","f","f","f","f","t","f","f","f","f","f","f","f","f","f","f","f","f","f"
"neg","f","f","f","f","f","f","t","f","f","f","f","f","f","f","f","f","f","f","f"
"pos","f","f","f","f","f","f","f","t","f","f","f","f","f","f","f","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","t","f","f","f","f","f","f","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","t","f","f","f","f","f","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","t","f","f","f","f","f","f","f","f"
"pos","f","f","f","f","f","f","f","f","f","f","f","t","f","f","f","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","t","f","f","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","f","t","f","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","f","f","t","f","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","t","f","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","t","f","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","t","f"
"neg","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","f","t"
```

nums3_1rel.arff

```
@relation 1-ary
@attribute rel-types
{"even","odd","prime"}
@attribute a
{"0","1","2","3","4","5","6","7","8",
"9","10","11","12","13","14","15","16
","17","18"}
@data
"even","0"
"even","2"
"even","4"
"even","6"
"even","8"
"even","10"
"even","12"
"even","14"
"even","16"
"even","18"
"odd","1"
"odd","3"
"odd","5"
"odd","7"
"odd","9"
"odd","11"
"odd","13"
"odd","15"
"odd","17"
"prime","2"
"prime","3"
"prime","5"
"prime","7"
"prime","11"
"prime","13"
"prime","17"
```

nums3_2rel.arff

```
@relation 2-ary
@attribute rel-types {"less_than"}
@attribute a
{"0","1","2","3","4","5","6","7","8",
"9","10","11","12","13","14","15","16
","17","18"}
@attribute b
{"0","1","2","3","4","5","6","7","8",
"9","10","11","12","13","14","15","16
","17","18"}
@data
"less_than","0","1"
"less_than","0","2"
"less_than","0","3"
"less_than","0","4"
"less_than","0","5"
"less_than","0","6"
"less_than","0","7"
"less_than","0","8"
"less_than","0","9"
"less_than","0","10"
"less_than","0","11"
"less_than","0","12"
"less_than","1","2"
"less_than","1","3"
"less_than","1","4"
"less_than","1","5"
"less_than","1","6"
"less_than","1","7"
"less_than","1","8"
"less_than","1","9"
"less_than","1","10"
"less_than","1","11"
"less_than","1","12"
"less_than","2","3"
"less_than","2","4"
"less_than","2","5"
"less_than","2","6"
"less_than","2","7"
"less_than","2","8"
"less_than","2","9"
"less_than","2","10"
"less_than","2","11"
"less_than","2","12"
"less_than","3","4"
"less_than","3","5"
"less_than","3","6"
"less_than","3","7"
"less_than","3","8"
"less_than","3","9"
"less_than","3","10"
"less_than","3","11"
"less_than","3","12"
"less_than","4","5"
"less_than","4","6"
"less_than","4","7"
"less_than","4","8"
"less_than","4","9"
"less_than","4","10"
"less_than","4","11"
"less_than","4","12"
"less_than","5","6"
"less_than","5","7"
"less_than","5","8"
"less_than","5","9"
"less_than","5","10"
"less_than","5","11"
"less_than","5","12"
"less_than","6","7"
"less_than","6","8"
"less_than","6","9"
"less_than","6","10"
"less_than","6","11"
"less_than","6","12"
"less_than","7","8"
"less_than","7","9"
"less_than","7","10"
"less_than","7","11"
"less_than","7","12"
"less_than","8","9"
"less_than","8","10"
"less_than","8","11"
"less_than","8","12"
"less_than","9","10"
"less_than","9","11"
"less_than","9","12"
"less_than","10","11"
"less_than","10","12"
```

```
"less_than","11","12"
"less_than","0","13"
"less_than","1","13"
"less_than","2","13"
"less_than","3","13"
"less_than","4","13"
"less_than","5","13"
"less_than","6","13"
"less_than","7","13"
"less_than","8","13"
"less_than","9","13"
"less_than","10","13"
"less_than","11","13"
"less_than","12","13"
"less_than","0","14"
"less_than","1","14"
"less_than","2","14"
"less_than","3","14"
"less_than","4","14"
"less_than","5","14"
"less_than","6","14"
"less_than","7","14"
"less_than","8","14"
"less_than","9","14"
"less_than","10","14"
"less_than","11","14"
"less_than","12","14"
"less_than","13","14"
"less_than","0","15"
"less_than","1","15"
"less_than","2","15"
"less_than","3","15"
"less_than","4","15"
"less_than","5","15"
"less_than","6","15"
"less_than","7","15"
"less_than","8","15"
"less_than","9","15"
"less_than","10","15"
"less_than","11","15"
"less_than","12","15"
"less_than","13","15"
"less_than","14","15"
"less_than","0","16"
"less_than","1","16"
"less_than","2","16"
"less_than","3","16"
"less_than","4","16"
"less_than","5","16"
"less_than","6","16"
"less_than","7","16"
"less_than","8","16"
"less_than","9","16"
"less_than","10","16"
"less_than","11","16"
"less_than","12","16"
"less_than","13","16"
"less_than","14","16"
"less_than","15","16"
"less_than","0","17"
"less_than","1","17"
"less_than","2","17"
"less_than","3","17"
"less_than","4","17"
"less_than","5","17"
"less_than","6","17"
"less_than","7","17"
"less_than","8","17"
"less_than","9","17"
"less_than","10","17"
"less_than","11","17"
"less_than","12","17"
"less_than","13","17"
"less_than","14","17"
"less_than","15","17"
"less_than","16","17"
"less_than","0","18"
"less_than","1","18"
"less_than","2","18"
"less_than","3","18"
"less_than","4","18"
"less_than","5","18"
"less_than","6","18"
"less_than","7","18"
"less_than","8","18"
"less_than","9","18"
"less_than","10","18"
"less_than","11","18"
"less_than","12","18"
"less_than","13","18"
"less_than","14","18"
"less_than","15","18"
"less_than","16","18"
"less_than","17","18"
```