EXPLOITING REDUNDANCY TO IMPROVE MACHINE LEARNING

by

Jeffrey Alan Lorenzen

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

December 1999

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Jeffrey Alan Lorenzen

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Ellen Riloff
	Gary Lindstrom
	Joseph Zachary

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Jeffrey Alan Lorenzen in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ellen Riloff
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This thesis introduces Recovery, a redundant rule learning system that provides improved accuracy over a number of widely used machine learning systems. Many machine learning techniques take a divide-and-conquer approach to learning whereby some part of the problem is solved, and then focus turns to the remaining, unsolved portion. Although this approach often leads quickly to concise solutions, it ignores the opportunity to add robustness. Recovery exploits the data by searching for additional, redundant rules that can add robustness and improve predictive accuracy. In addition to redundant rules, Recovery also handles missing values differently than most systems, namely by ignoring them both during rule generation and classification. Finally, Recovery introduces a new rule weighting scheme called vector magnitude that contributes to the improved results. This thesis describes Recovery and shows results comparing it to CN2, RIPPER, and C4.5. Recovery generally achieves better performance across a variety of learning tasks and is worthy of consideration when selecting a learning system.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Redundancy is an effective way to improve robustness. When communicating with others it is common to use repetition, often slightly reworded, to reinforce a point or avoid misunderstanding. Redundancy in telephone networks allows calls to be routed even when a link is down. Airplanes are designed with redundant systems to ensure that, even if one system fails, the plane will still fly.

This thesis shows that machine learning can benefit from redundancy. In particular, this thesis demonstrates that exploiting redundancy improves the performance of concept classifiers by adding robustness. A redundant learning algorithm called Recovery is presented and evaluated against several other learning systems including CN2 [3], RIPPER [4], and C4.5 [12]. The results from testing the systems on ten data sets covering a variety of learning tasks are presented and show that Recovery generally outperforms the other systems. Results also demonstrate that ignoring missing values when learning rules and using them for classification dramatically improves results. A new metric called *vector magnitude* is introduced and its usefulness when constructing learners is demonstrated.

1.1 Concept Classifiers

A system can be said to learn if it improves as a result of experience [9]. Machine learning is concerned with developing computer programs that exhibit this type of improvement. There are many tasks that fall under the heading of machine learning including programs that learn to play checkers, systems that learn to recognize handwriting, and robots that learn to drive.

This thesis focuses on one particular machine learning task: building concept classifiers. Concept classifiers take a set of examples that typify a concept or set of concepts. From these examples, concept classifiers learn a set of patterns that capture the concept or concepts. These patterns can then be used to classify new examples that the system has not seen before. Concept classifiers have been used successfully in a variety of learning

applications including medical diagnosis, determination of consumer credit risk, and text classification.

1.2 Common Approaches to Building Concept Classifiers

Concept classifiers can be built in a number of ways. Two widely used techniques for building classifiers are covering algorithms and decision trees. This section briefly describes the two techniques. Adding redundancy to classifiers will be discussed in Section 1.3.

1.2.1 Covering Algorithms

One popular method for creating classification systems is covering algorithms. Given a set of examples as experience, covering algorithms create a list of IF ... THEN ... rules that classify the examples.

Covering algorithms take a straightforward divide-and-conquer approach to learning. These algorithms start by finding a rule that correctly classifies some of the examples. This newly created rule is added to the rule list and the examples that it classifies (covers) are removed. The process is repeated with the remaining examples until all the examples are covered. A basic covering algorithm is given in Figure 1.1.

The basic covering algorithm generates an ordered list of rules. To classify a new, previously unseen instance, the rule list is searched in order. The new instance is checked against each rule to see if the rule applies. If so, the classification given by the rule is assigned to the new instance and the search is halted. If no rule matches, a default classification can be assigned.

```
let RULE_LIST be empty
let EXAMPLES be the set of all training examples

repeat {
   select the best rule and add it to RULE_LIST
   remove EXAMPLES covered by best rule
} until EXAMPLES is empty
```

Figure 1.1. A basic covering algorithm

Not all covering algorithms generate ordered rule lists. In fact, there are many variants of the basic algorithm. All work by creating a rule that covers some of the training examples, removing those examples, and continuing until all examples are covered. An algorithm for creating unordered rule lists will be investigated later.

To illustrate the basic covering algorithm, consider a variant of the "play/don't play" data set used by Quinlan [12]. Figure 1.2 shows information about the weather on a particular day and whether or not it was a good one to play golf. Each line of information is an example to the learning system. The goal is to create a system that can predict whether it is a good day to play golf given information about the outlook, temperature, humidity, and wind. A correct answer is one that is consistent with the training examples given.

In examining the data from the play/don't play data set, notice that on each line where `outlook = "overcast"` that `play = "yes"`. The basic covering algorithm[1] would add a rule:

```
IF (outlook = "overcast") THEN play = "yes"
```

[1]Laplace accuracy (described in Section 3.1.2) is used to select the best rule but this detail is unimportant for illustration purposes.

outlook	temperature	humidity	windy?	play?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rain	mild	high	false	yes
rain	cool	normal	false	yes
rain	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rain	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rain	mild	high	true	no

Figure 1.2. Play/don't play data set

Then, the four examples covered by that rule would be removed from further consideration. The algorithm would continue trying to find rules to classify the remaining ten examples. It might notice that when `outlook = "rain"` and `windy = "false"` that `play = "yes"`. The following rule capturing this would be added.

```
IF ((outlook = "rain") AND (windy = "false")) THEN play = "yes"
```

The three examples covered by this rule would be removed and the process would repeat on the remaining seven examples. Next, the following rule would be created.

```
IF (humidity = "high") THEN play = "no"
```

This rule covers four examples, leaving three uncovered. The next rule

```
IF (outlook = "sunny") THEN play = "yes"
```

would catch two of those. The remaining uncovered example would be caught by adding the following rule:

```
IF (outlook = "rain") THEN play = "no"
```

The complete list of rules created by the basic covering algorithm is shown in Figure 1.3. The numbers to the right are line numbers for reference. This list of rules captures the essence of a good day to play golf, at least according to the training examples that were given. When presented with information about a new day, the system would search this list of rules, in order, and return the class of the first rule that matches.

```
IF (outlook = "overcast") THEN play = "yes"                          (1)
IF ((outlook = "rain") AND (windy = "false")) THEN play = "yes"      (2)
IF (humidity = "high") THEN play = "no"                              (3)
IF (outlook = "sunny") THEN play = "yes"                             (4)
IF (outlook = "rain") THEN play = "no"                               (5)
```

Figure 1.3. Rules created by a basic covering algorithm for the play/don't play data

1.2.2 Decision Trees

Another method for building concept classifiers is decision trees. As with covering algorithms, decision trees take a divide-and-conquer approach to learning. A decision tree algorithm starts by considering the full set of training examples available, but instead of selecting the best attribute-value pairs (such as `outlook = "sunny"`) as done in a covering algorithm, a decision tree algorithm selects the best attribute (perhaps `outlook`) and creates a branch for each possible value of that attribute (`"sunny"`, `"overcast"`, `"rain"`). Examples are passed down the appropriate branch based on their value for the attribute being tested. If all examples at a child node belong to the same class, that node becomes a leaf node that returns the class held by the examples in it. Otherwise, the procedure is repeated recursively for each new child node, considering only the examples at that node. A basic decision tree algorithm is shown in Figure 1.4.

Classification of new instances with decision trees starts at the root node. The value in the new instance for the attribute at the root node is used to decide which branch to traverse. If the child node reached is a leaf node, it provides the classification to assign to the new instance. If the child node is not a leaf, then the attribute stored there is checked and a new branch traversed. This process is repeated until a leaf node is reached.

This algorithm is best illustrated with an example. Consider the "play/don't play" data set listed in Figure 1.2. The basic decision tree algorithm would create the decision tree shown in Figure 1.5.

```
let NODE be an empty root node
let EXAMPLES be the set of all training examples

if all EXAMPLES at NODE belong to the same class then {
   make NODE a leaf node return that class
} else {
   select the best attribute to branch on
   assign the selected attribute to NODE
   create a branch for each possible value of the attribute
   send EXAMPLES down branches based on value
   repeat for the EXAMPLES at each new child NODE
}
```

Figure 1.4. A basic decision tree algorithm

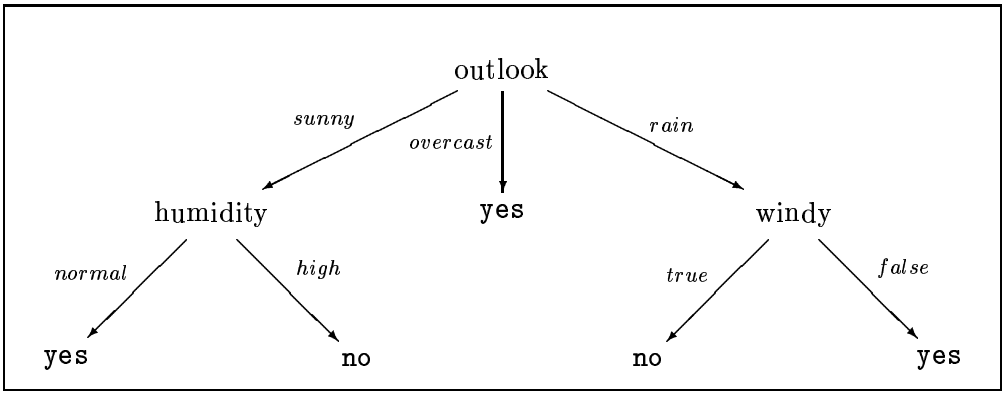

Figure 1.5. A basic decision tree created for the play/don't play data

Both covering algorithms and decision trees are widely used techniques for building concept classifiers. Both algorithms are relatively easy to understand and both produce concise classifiers. In their effort to build a concise solution, however, both techniques ignore the opportunity to benefit from other good patterns in the data. The next section proposes ways that redundancy could add robustness to these techniques.

1.3 Adding Redundancy to Classifiers

The problem with the divide-and-conquer approaches to learning lies in the attempt to move quickly toward a concise solution. Most algorithms, including covering algorithms and decision trees, find the "best" solution to part of a problem then use that to divide the problem so they can focus on the unsolved portion. By only considering the single "best" solution, these algorithms ignore other patterns that could be used to add robustness.

To improve the robustness of covering algorithms, we propose making better use of the data. Rather than just selecting the single best pattern and using it to partition the data, we propose selecting a set of good patterns and possibly using them all as part of the solution.

In the case of covering algorithms (we focus on these for simplicity), we propose adding redundant rules. Rather than just selecting a single rule and using it to partition the data, we propose selecting a set of good rules and evaluating them all. Each may be kept in the final rule set if it is likely to help classification. A redundant covering algorithm is shown in Figure 1.6.

The term *redundancy* is used because multiple rules are being selected in an effort

```
let RULE_LIST be empty
let EXAMPLES be the set of all training examples

repeat {
   select the best rule and add it to RULE_LIST
   select additional highly accurate, highly applicable rules
      and add them to RULE_LIST
   remove EXAMPLES covered by all rules that were added
} until EXAMPLES is empty
```

Figure 1.6. A basic redundant covering algorithm

to cover the training examples more than one time. The goal is to increase robustness in the system through this redundancy. Even though a single example may get covered multiple times in an ordinary covering algorithm, it is not intentional. Here, redundancy is exploited by intentionally keeping rules that may provide multiple coverage of examples provided that the rules are highly accurate and apply to many examples. Redundancy could be exploited in other machine learning techniques such as decision trees to increase their robustness as well.

This thesis makes the following claim:

Exploiting redundancy can improve the predictive accuracy of conceptual classifiers.

The rest of the thesis is devoted to substantiating this claim. In the next section, the claim is motivated by showing examples where an ordinary covering algorithm fails and how redundancy could possibly help. In Chapter 3, details of the redundant covering algorithm are given. Chapter 4 presents results of experimenting with the redundant learning system and shows that redundancy does indeed improve accuracy.

1.4 A Motivating Example

The claim made in this thesis can be motivated by considering a synthetic data set. This data set was contrived to highlight cases where ordinary covering algorithms can fail and to show how redundancy could alleviate some of the problems.

The synthetic data set consists of two attributes and two possible classes. The first attribute, `a1` has three possible values: `"v1"`, `"v2"`, and `"v3"`. The second attribute `a2` also has three possible values: `"v4"`, `"v5"`, and `"v6"`. The two possible classes are `"+"` and `"-"`. This data set is shown in Figure 1.7.

The basic covering algorithm listed in Figure 1.1 would generate the two rules shown in Figure 1.8. For simplicity, precision was used to select the "best" rule where *precision* is calculated by taking the number of training instances classified correctly divided by the total number of training instances to which the rule applies. Both rules have a precision of 100%.

While these two rules are sufficient to correctly classify all of the training instances shown in Figure 1.7, there are many possible examples that would not be classified correctly. Some of these cases will be explored in the next two section.

a1	a2	class
v1	v4	+
v1	v4	+
v1	v4	+
v1	v4	+
v1	v5	+
v1	v5	+
v1	v5	+
v1	v5	+
v2	v6	-
v2	v6	-
v3	v6	-
v3	v6	-

Figure 1.7. A synthetic data set

```
IF (a1 = "v1") THEN class = "+"   (1)
IF (a2 = "v6") THEN class = "-"   (2)
```

Figure 1.8. Rules generated by a basic covering algorithm for the synthetic data

1.4.1 Missing Values

Missing values occur in the data when the value for an attribute is not available. Perhaps a sensor malfunctioned and did not record a value or maybe obtaining the value required an expensive or dangerous medical test that the doctors opted not to perform. In any case, missing values are a real world problem.

The two rules in Figure 1.8 would not correctly classify a new instance where `a1 = "v2"` and attribute `a2` is undefined. This is a case with a missing value. Notice that all examples that have the value `"v2"` for attribute `a1` belong to class `"-"`, but the covering algorithm does not produce any rule that would lead to this classification.

Some systems deal with the problem of undefined values by assigning the majority value for that attribute in place of the missing values, but here, there is no majority value: each of the possible values (`"v4"`, `"v5"`, and `"v6"`) occur with equal frequency. One of these values would have to be chosen arbitrarily. This covering algorithm would still have a two-thirds chance of misclassifying that instance.

1.4.2 Previously Unseen Values

Another real world problem is previously unseen values. This may be an artifact of sparse training data – the value exists but was not seen in training – or it may be a new value – the original survey only included `"blonde"`, `"brown"`, `"black"`, and `"red"` for the attribute `hair-color` but a person with white hair responded. Either way, previously unseen values can occur in the data and must be handled.

Consider a new instance where `a1 = "v2"` and attribute `a2` has the previously unseen value of `"v7"`. The system would not be able to classify the instance because no rules apply. As with the previous example, the value of `"v2"` for attribute `a1` is a strong indicator of class `"-"` but the algorithm did not produce a rule that would lead to this conclusion.

1.4.3 Solution: Redundancy

Both of the above problems could be solved if the redundant rules listed in Figure 1.9 were added to the system. Note that each rule has a precision of 100% on the training data: each rule would correctly classify every training case to which it applies.

For the cases shown in Section 1.4.1 and 1.4.2, not all of these rules are necessary. In fact, only the fifth rule (5) would be necessary to classify both examples correctly. All the others are perfect rules too, but it is questionable as to whether or not we should

```
IF (a2 = "v4") THEN class = "+"                          (1)
IF (a2 = "v5") THEN class = "+"                          (2)
IF ((a1 = "v1") AND (a2 = "v4")) THEN class = "+"        (3)
IF ((a1 = "v1") AND (a2 = "v5")) THEN class = "+"        (4)
IF (a1 = "v2") THEN class = "-"                          (5)
IF (a1 = "v3") THEN class = "-"                          (6)
IF ((a1 = "v2") AND (a2 = "v6")) THEN class = "-"        (7)
IF ((a1 = "v3") AND (a2 = "v6")) THEN class = "-"        (8)
```

Figure 1.9. Possible redundant rules for the synthetic data

create and retain all of them because of the abundance of redundant rules that could be introduced to the system.

This section has illustrated that redundancy has the potential to succeed. It has presented two types of examples that could benefit from redundancy: missing values and previously unseen values. The goal, however, is to increase the overall performance of learning systems, regardless of the source of the improvements.

1.5 Summary

The goal of this thesis is to show that exploiting redundancy can improve machine learning. This chapter has introduced ideas and presented an example that motivates the work. Chapter 2 provides background from the field of machine learning and describes related work in building classifiers and using redundancy. Chapter 3 presents the redundant covering algorithm Recovery. Chapter 4 shows the results of comparing Recovery to other widely used systems. The final chapter provides conclusions drawn from this work with redundancy and discusses directions for future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides the background for describing the redundant rule learner Recovery. The chapter begins with a review of some machine learning terminology that will be used throughout the thesis. Next, related work in the field is described. The information presented in this chapter serves as a foundation for explaining how Recovery exploits redundancy.

2.1 Machine Learning

As mentioned in Chapter 1, the goal of machine learning is to create computer programs that improve with experience [9] where improvement is defined as a measurable increase in some aspect of performance. The improvement is typically measured as *accuracy*, a measure of how well the system has learned what it was supposed to learn. The experience is the information provided to the system, from which learning takes place. This experience may come in a variety of forms, such as databases of knowledge or transcripts of sessions with users.

This thesis focuses on techniques where the experience is in the form of *examples* or *instances*. An example or instance is a description of a particular entity or process. The examples from which learning takes place are referred to as the *training data*. The new examples over which the classifier is evaluated are referred to as the *test data*.

Often examples are specified in terms of *attributes* and *values*. An attribute is a field in the description of an example, perhaps `hair-color` or `weight`. Possible values for the attribute `hair-color` might include `"blonde"`, `"brown"`, and `"black"`. An *attribute-value pair* specifies both an attribute and one of its possible values, such as

```
hair-color = "brown"
```

The learner may be presented with all the experience initially and be expected to learn from it in a single pass (*batch learning*). Alternatively, new experience may be

presented over time and the learning can be a continuous process (*incremental learning*). While the latter may appear more useful, often batch learning is simpler and sufficient. Sometimes algorithms are developed to work in batch mode and later modified to perform incremental learning.

Machine learning algorithms can also be characterized by the supervision they require. *Supervised learning* is learning where the examples must be preclassified before learning can take place. This may require a human to review the examples and assign class labels to each of them. *Unsupervised learning* does not require the data to be preclassified. The benefit of supervised learning is that the classification can be leveraged by the learning technique to evaluate itself as it goes. Covering algorithms and decision trees are examples of supervised learning. Conceptual clustering algorithms, which group or "cluster" similar examples based on attribute values without knowing the correct classification for each example, perform unsupervised learning.

2.2 Concept Classifiers

Concept classifiers or "classification systems" are machine learning programs that start from a set of training examples and produce a *concept description*. A concept description is a representation of the knowledge learned. From the training data, the algorithm attempts to learn patterns that typify this concept. This description can then be used to classify new examples.

2.3 Covering Algorithms

Covering algorithms are a supervised machine learning technique for building classification systems. From the preclassified training data, they construct a list of rules that can be used to classify new instances that have not been seen before. A basic covering algorithm was given in Figure 1.1.

Rules are generally of the form:

$$\texttt{IF} < \textit{condition} > \texttt{THEN class} = \textit{class}_x$$

where

$< \textit{condition} >$ is a boolean expression, usually confined to the conjunct of attribute-value pairs

$class_x$ is one of the possible classifications or categories

A sample rule might be

```
IF ((hair-color = "brown") AND (age = "27")) THEN class = "nice-guy"
```

The lists of rules created by covering algorithms may be either *ordered* or *unordered*. The former are simpler to produce, but are more difficult for humans to interpret because each rule can only be considered in the context of the rules that precede it. Unordered rule lists (sometimes referred to as *rule sets*) are simpler to understand because each rule can stand independent of the others.

Covering algorithms may perform either batch or incremental learning. This thesis focuses on those that perform batch learning because this form of learning is simpler and more common.

2.3.1 AQ

One of the earliest covering algorithms was AQ [7]. It worked by selecting a seed example from the training data and using it to build a rule that covered the selected example and possibly others of the same class so long as it covered *no* examples of any other class. It added the new rule to the rule list and removed the seed example and all others to which the rule applied. This process was repeated until all examples were covered. In this manner, an unordered set of rules was created.

There were several variants of AQ, including AQ15 [8], which performed incremental learning, but none exploited redundancy.

2.3.2 CN2

CN2 [3] improved upon AQ by eliminating the need for a seed example. It also relaxed the constraint that the rules generated must correctly cover all examples to which they applied. This allowed the algorithm to work well in cases when the training data were *noisy*, that is the data were not always perfect. Despite these improvements, CN2 still made no use of redundancy.

CN2 performs a search for the "best" rule, one that classifies many examples correctly and as few as possible incorrectly. It adds this rule to the rule list, removes the covered examples and repeats until no more good rules can be found. This results in an ordered list of rules.

Improvements to CN2 [2] changed the way candidate rules were evaluated and gave the option of generating either ordered or unordered rule lists. The new version uses Laplace accuracy instead of entropy (both metrics will be described in Section 3.1.2) to evaluate conditions when searching for the best rule. Chapter 3 described CN2 in greater detail since CN2 is the basis for the redundant rule learning system described in this thesis.

2.3.3 RIPPER

RIPPER [4] is another covering algorithm. Its advantages include that it is faster than many other covering algorithms, especially on large, noisy data sets. It has an especially efficient pruning algorithm that can reduce the problem of overfitting (creating rules that are too specific to the training examples). RIPPER generally performs well, frequently beating C4.5rules (described later) on a variety of data sets. RIPPER does not exploit redundancy.

The current version of RIPPER allows *set-valued features* [5]. These allow features that frequently have no value to be represented efficiently. This is particularly useful in text classification, where the features are often the words in the vocabulary and any given text contains only a small subset of the possible values.

2.4 Redundant Covering Algorithms

2.4.1 Cestnik and Bratko

The idea of exploiting redundancy in machine learning was first explored by Cestnik and Bratko [1]. Their system exhaustively explored all combinations of up to two attributes. All "good" rules were retained. A good rule was one that was perfect over the examples it covered and was not a specialization of a rule that was already in the rule set.

Results from their system showed that adding redundancy to rule learning systems produced classifiers with higher accuracy. Unfortunately, the exhaustive search required them to limit the rules to combinations of two attributes, possibly preventing them from finding other good rules. Even with a limit of two attributes, their algorithm generated an order of magnitude more rules than an ordinary covering algorithm.

2.4.2 DAIRY

More recent work on adding redundancy to covering algorithms was done by Hsu, Etzioni, and Soderland with their system called DAIRY [6]. They started from an unordered covering algorithm and proposed two extensions. The first, "recycling," does not remove examples covered by a rule but considers them at a reduced weight in later iterations of the algorithm. Recycling improved performance in all their test cases and appears to be a win. The second extension they proposed was a second iteration through the training data performing a massive search for completely redundant rules. These rules do not cover any previously uncovered examples but are still good according to user specified bounds on accuracy, coverage, and depth. The results of using completely redundant rules were not as conclusive but warranted further exploration.

The work done with Recovery is similar to that done with DAIRY but differs in several key ways. First, Recovery does not consider the covered examples at a discounted weight because examples do not lose any significance just because they have been covered once. The problem with arbitrarily generating redundant rules is that there may be so many good redundant rules that some examples never get covered because rules that cover them are weaker than redundant rules that apply to more cases. DAIRY addresses this problem by reducing the weight of covered examples at each step so that the uncovered examples will eventually have enough weight that rules to cover them are created. Recovery avoids this discounted weighting scheme.

Another difference between Recovery and DAIRY is that DAIRY imposed a limit of two attribute-value pairs when building rules. Recovery avoids this artificial limit by carefully pruning the search space. The approach taken by Recovery is more flexible in that it can consider rules with more than two attributes.

The authors of DAIRY did not evaluate the effects of the redundant rules alone, without "recycling." These effects are investigated with Recovery and show that redundant rules are more valuable than the work with DAIRY illustrated.

2.5 Decision Trees

Another approach to building classification systems is to build decision trees. Whereas covering algorithms select attribute-value pairs as the basis for partitioning the data, decision trees work by selecting an attribute on which to branch, then creating branches for each possible value of that attribute. The basic algorithm for constructing decision

trees is relatively straightforward, and classifying new instances using these trees is simple because there is a unique path through the tree for any specific example. A basic decision tree algorithm was given in Figure 1.4.

2.5.1 C4.5

C4.5 [12] is a popular decision tree implementation. Its predecessor, ID3 [11], used *information gain* to select the best attribute to branch on. Information gain, unfortunately, is biased toward attributes that have many values. C4.5 overcame this by using a slightly different evaluation criteria called *gain ratio*.

C4.5rules can be used to convert a decision tree into a list of production rules. These rule lists are similar to those produced by covering algorithms, but the procedure for generating them is less intuitive and the results are generally not as good.

2.5.2 ID5

ID5 [14] is another decision tree algorithm. It is similar to ID3, but performs incremental learning. It can process new training instances over time by revising the decision tree rather than building a completely new tree. This allows the decision tree to evolve over time as the training data changes.

An important thing to note about decision trees is that each example must have a value for each attribute or one must be provided. Consider the task of classifying an example containing a missing value for a particular attribute. If a node for this attribute is encountered in the tree, the branch to take is undetermined. C4.5 handles this by substituting the most frequent value in places of missing values. Results presented later show that this can be a problem. Recovery takes a different approach and achieves better results.

CHAPTER 3

A REDUNDANT COVERING ALGORITHM

This chapter explains our approach to adding redundancy to covering algorithms. The resulting system is called Recovery. The first section gives a thorough description of a nonredundant covering algorithm since that is the stating point for this work. Then, the modifications made to create the redundant rule learning system Recovery are described.

Recovery is built upon a nonredundant covering algorithm similar to CN2 [3]. This chapter describes the three major modifications made to the CN2 algorithm to create Recovery. The first major change adds redundant rules to the rule lists. Another change involved modifying the way rules are evaluated when the examples they cover contain missing values. Recovery ignores missing values rather than substituting values for them. The final change made in Recovery is the use of a new metric called vector magnitude to weight the rules during classification. This new metric helps alleviate the problems of rules that are highly accurate but only cover a few examples.

3.1 A Nonredundant Rule Learner

As mentioned in Chapter 1, covering algorithms can be used to produce either ordered or unordered lists of rules. A basic ordered covering algorithm was presented for simplicity. Because Recovery creates unordered rule lists, the remainder of this thesis focuses on unordered rules. The procedure for creating unordered rule sets is largely the same as that for producing ordered rules but the way the rules are applied is somewhat different.

At the heart of Recovery is a nonredundant rule learning system similar to CN2. Code to incorporate redundancy was added to this nonredundant system. A thorough understanding of the nonredundant system will make it easier to describe the changes made. The algorithm used in CN2 is described in the next section. Section 3.2 describes the modifications made to create Recovery.

3.1.1 Nonredundant Rule Generation with CN2

CN2 has the option of creating either ordered or unordered rule lists. Unordered rule lists are easier for humans to interpret and the performance when classifying new instances is generally better [2]. The default in CN2 is to create unordered rules and that is the algorithm described here.

The CN2 algorithm for generating unordered rules begins with a procedure called `CN2Unordered`. This procedure takes the set of all training examples and a list of the possible classes for the data set and iterates through these possible classes, treating each in turn as the "target class." For each target class, `CN2Unordered` calls `CN2ForOneClass` to generate rules to cover examples of that class. The rules returned are added to the final rule list. This procedure is listed in Figure 3.1.

`CN2ForOneClass` is given a class and the set of all training examples. It returns a list of rules that cover the examples belonging to CLASS. It creates this list by repeatedly calling `CN2FindBestCondition` to find the "best" condition for the given class. Remember that a condition is a conjunct of attribute-value pairs. For each condition returned, a rule of the form `IF (`*condition*`) THEN CLASS` is created and added to the set of rules to be returned. Next, the examples covered by the new rule are removed. The process is then repeated with the remaining training examples. This stops when all the examples are covered or no good rule can be found to cover the remaining examples. `CN2ForOneClass` is shown in Figure 3.2.

Note that only the examples that are covered *and have the same CLASS as the new rule* are removed. Examples that belong to other classes are retained so that a rule that

```
procedure CN2Unordered( ALLEXAMPLES, ALLCLASSES ) {
   let RULESET = {}
   for each class CLASS in ALLCLASSES {
      call CN2ForOneClass(ALLEXAMPLES, CLASS) to generate RULES
      add RULES to RULESET
   }
   return RULESET
}
```

Figure 3.1. The CN2 algorithm for generating unordered rules

```
procedure CN2ForOneClass( EXAMPLES, CLASS ) {
   let RULES = {}
   repeat {
      find BESTCOND by calling CN2FindBestCondition( EXAMPLES )
      if BESTCOND is not null then {
         add to RULES the rule "if BESTCOND then predict CLASS"
         remove from EXAMPLES all those in CLASS covered by BESTCOND
      }
   } until BESTCOND is null
   return RULES
}
```

Figure 3.2. The CN2 algorithm for covering one class

correctly classifies them may be found when the class they belong to is the target class.

The best rule is one that correctly classifies as many examples as possible while misclassifying as few as possible. To find the condition for this rule, CN2 employs a beam search. A beam search is a modified best-first search, but instead of exploring only the single best path at any given point, a small number of the most promising alternatives are explored. The goal of using a beam search is to alleviate the problem of local maxima.

The procedure that implements the beam search is called `CN2FindBestCondition`[1]. The search starts with a condition that contains only the value "true." This condition matches all examples. This initial condition is added to the collection of conditions being explored (referred to as `STAR` in the algorithm and corresponding to the beam in the search). `CN2FindBestCondition` repeatedly selects a condition from STAR and expands it by adding an attribute-value pair that is not already in the condition. The new condition (COND1) is evaluated over the remaining examples of CLASS to which it applies using Laplace accuracy (described below). If this new condition improves upon the best condition seen so far (BESTCOND), then the new condition is retained as BESTCOND. Additionally, if this new condition beats any of the conditions in STAR (or the size of STAR is less than the beam width), this new condition is added to STAR and STAR is trimmed to the beam width (referred to as MAXSTAR in the algorithm). The difference between BESTCOND and the conditions in STAR is that BESTCOND is

[1]The test for significance is omitted.

the single best condition (the one with the highest Laplace accuracy) that has been seen thus far and STAR contains conditions that may not be as good, but have the potential to be better. The potential of a rule is the maximum Laplace accuracy it could achieve if it covered only examples from CLASS and none from any other class. The main loop is then repeated until no condition better than BESTCOND can be found. At the end of the loop, the single best condition is returned. This procedure is shown in Figure 3.3.

In this manner, all possible attribute-value combinations will be tried as candidate conditions for the rule, provided they improve upon other conditions already seen. The beam search provides a way to explore more of the search space than a greedy, best-first search, but avoids the expense of a fully exhaustive search.

3.1.2 Rule Evaluation Metrics

The original CN2 used a measure called entropy to evaluate rules. Entropy captures the accuracy of a rule and returns higher values for rules that cover examples of only a

```
procedure CN2FindBestCondition( EXAMPLES ) {
   let STAR initially contain only the most general condition ("true")
   let NEWSTAR = {}
   let BESTCOND = null
   while STAR is not empty {
      for each condition COND in STAR {
         for each possible attribute test not already in COND {
            let COND1 = COND & TEST
            evaluate COND1 over EXAMPLES using LaplaceAccuracy
            if COND1 is better than BESTCOND
               then let BESTCOND = COND1
            add COND1 to NEWSTAR
            if size of NEWSTAR > MAXSTAR (a user-defined constant)
               then remove the worst condition in NEWSTAR
         }
      }
      let STAR = NEWSTAR
   }
   return BESTCOND
}
```

Figure 3.3. The CN2 algorithm for finding the best condition

single class. Entropy is given by the following formula:

$$\texttt{Entropy} = -\sum_c((n_c/n_{tot}) * \log_2(n_c/n_{tot}))$$

where

n_c is the number of examples covered by the rule in the predicted class c

n_{tot} is the total number of examples covered by the rule

The problem with entropy is that it tends to select rules that cover only a few examples because those rules are more likely to have higher accuracy. This bias is undesirable since their high accuracy on the training data may not reflect their performance on new instances.

In the improved version of CN2 [2], a new metric is used to evaluate candidate conditions. This metric is called Laplace accuracy and is computed as follows.

$$\texttt{LaplaceAccuracy} = (n_c + 1)/(n_{tot} + k)$$

where

k is the number of classes in the domain

n_c is the number of examples covered by the rule in the predicted class c

n_{tot} is the total number of examples covered by the rule

`LaplaceAccuracy` returns a value between 0 and 1, with higher values being better. It is a useful metric because it captures the precision of a rule (n_c/n_{tot}) and takes into account the number of possible classes.

Consider an instance from the play/don't play data given in Figure 1.2. One of the rules that was evaluated during rule generation was

```
IF (outlook = "overcast") THEN play = "yes" [2 3]
```

The numbers in brackets indicate the class distribution of the examples covered by this rule: there were 2 examples belonging to class `play = "yes"` and 3 belonging to class `play = "no"`. Since there are two classes, k would be 2. The target class is `play = "yes"`, so $n_{play=\text{“}yes''}$ would be 2, n_{tot} would be 5 and `LaplaceAccuracy` would evaluate

to 0.4286. Note, this rule does not appear in the final rule list because other rules with better values for `LaplaceAccuracy` were discovered.

A shortfall of `LaplaceAccuracy` is that it does not consider the distribution of the examples not belonging to the class c. Take a classification task with three possible classes. Consider two rules, one that partitions 20 examples as follows: 10 in $class_1$, 9 in $class_2$, and 1 in $class_3$ and another rule that partitions them as: 10 in $class_1$, 5 in $class_2$, 5 in $class_3$. Since there are 20 total examples, $n_{tot} = 20$. There are three classes, so $k = 3$. For a target class of $class_1$, $n_{class_1} = 10$, and `LaplaceAccuracy` returns 0.4783 for both rules. We feel the first rule is superior because adding another attribute-value pair to its condition gives a better chance of isolating the 9 examples that belong to $class_2$, rather than leaving a perfect 5/5 split between $class_2$ and $class_3$. A metric that takes into account the distribution of examples across classes is presented when Recovery is described in Section 3.2.2.

3.1.3 Classification with CN2

After generating a rule list, CN2 can be used to classify new instances. With an ordered rule list, it is sufficient to scan the list in order, returning the class of the first rule that matches. If no rule applies, then a default class is returned. With unordered rule lists, a mechanism is needed to resolve cases where more than one rule applies.

When classifying new instances with unordered rule lists CN2 uses a voting scheme to resolve cases where multiple rules apply. In CN2, the rules vote according to the distribution of training examples covered. These distributions are summed and the class with the most votes is returned.

Note that rules are evaluated differently during generation than during classification. During generation, rules are evaluated according to `LaplaceAccuracy` using the number of examples *remaining to be classified*. For example, if the first rule covered half the training examples, the next rule will be evaluated over only the remaining half of the examples. This is necessary to ensure that all examples of a class get covered. During classification, rules vote according to their distribution over the *entire set of training examples*. Using the total number of training examples covered gives a better indication of the true predictivity of the rule.

Consider the following example from the play/don't play data set given in Figure 1.2.

outlook	temperature	humidity	windy?	play?
rain	mild	high	false	yes

Rules (2) and (3) from Figure 1.3 both apply to the instance shown. These rules are listed again here, this time showing the class distribution of the training examples each rule covers.

```
IF ((outlook = "rain") AND (windy = "false")) THEN play = "yes"   [4 0]
IF (humidity = "high") THEN play = "no"                           [0 3]
```

The numbers in square brackets show the distribution of examples covered. For the first rule, [4 0] means the rule covered four training examples from class `play = "yes"` and zero examples from class `play = "no"`. Likewise, for the second rule, [0 3] means the rule covered zero examples from class `play = "yes"` and three examples from class `play = "no"`. CN2 would sum these distributions to get [4 3] and would return the class with the highest distribution, `play = "yes"`.

3.2 Recovery

This section describes the modifications made to CN2 to incorporate redundancy. Section 3.1 described the nonredundant covering algorithm used by CN2 in detail. This section introduces Recovery. It describes how redundant rules are generated and how they are added to the final rule list. This section also explains how Recovery handles missing values differently than many other learning systems and how Recovery weights the rules differently when using them for classification of new instances.

3.2.1 Redundant Rule Generation with Recovery

This section describes how Recovery generates redundant rules. Recovery is similar to CN2 but includes a number of modifications to generate and add redundant rules.

The procedure that drives the generation of redundant rules is the same as that used by CN2 except that the name has been changed from `CN2Unordered` to `RecoveryUnordered` and it now calls `RecoveryForOneClass` instead of `CN2ForOneClass`. As in CN2, this procedure iterates through the possible classes calling `RecoveryForOneClass` to generate rules for each one. It takes the rules returned and adds them to the final rule list. This new procedure is shown in Figure 3.4 with the changed lines marked by a '$\Leftarrow$'.

The first significant differences between CN2 and Recovery are noticeable in the procedure `RecoveryForOneClass`. Instead of calling `CN2ForBestCondition` to find only

```
procedure RecoveryUnordered( ALLEXAMPLES, ALLCLASSES ) {                    ⇐
   let RULESET = {}
   for each class CLASS in ALLCLASSES {
      call RecoveryForOneClass(ALLEXAMPLES, CLASS) to generate RULES        ⇐
      add RULES to RULESET
   }
   return RULESET
}
```

Figure 3.4. The Recovery algorithm for generating unordered rules

the single best condition for a rule, `RecoveryFindBestConditions` is called to find both BESTCOND and a set of other good conditions. As with CN2, a rule is created from BESTCOND and added to the rule list and the examples covered by this rule are removed. Recovery goes on to consider the conditions in OTHERGOODCONDS looking for more good conditions from which to create redundant rules. `RecoveryForOneClass` is listed in Figure 3.5.

To add redundant rules, `RecoveryForOneClass` considers the conditions in OTHERGOODCONDS. From each condition, a rule is constructed and `LaplaceAccuracy` is used to evaluate the rule. If the value of the new rule is within BESTCOND_SIMILARITY (a percentage) of the value of the rule created from BESTCOND, then the new rule is a candidate to be added to the rule list. For example, if the rule created from BESTCOND has a `LaplaceAccuracy` of 0.90 and BESTCOND_SIMILARITY is 0.95, then any rule with a `LaplaceAccuracy` of 0.90 * 0.95 = 0.855 or higher becomes a candidate rule. This threshold can be set by the user and controls how good the redundant rules must be to be considered. A value of 1.0 requires the redundant rules to be at least as good as the rule created from BESTCOND. Relaxing the threshold allows other "good" rules to be added but relaxing it too much would allow unreliable rules to enter the rule list.

There is no limit placed on the number of redundant rules that can be added but this number is affected by the value of MAXSTAR used to limit the beam search. The beam only holds conditions that have the potential to beat the current best condition. Any condition equivalent to the best condition is not placed in the beam. These conditions are placed in OTHERGOODCONDS. If there are multiple conditions as good as the

```
procedure RecoveryForOneClass( EXAMPLES, CLASS ) {                    ⇐
   let RULES = {}
   repeat {
      find BESTCOND and OTHERGOODCONDS by                             ⇐
         calling RecoveryFindBestConditions( EXAMPLES )               ⇐
      if BESTCOND is not null then {
         add to RULES the rule "if BESTCOND then predict CLASS"
         remove from EXAMPLES all those in CLASS covered by BESTCOND
         if this is the first iteration for CLASS then {              ⇐
            for each COND in OTHERGOODCONDS {                         ⇐
               if COND beats threshold and is not subsumed then {     ⇐
                  add to RULES the rule "if COND then predict CLASS"  ⇐
               }                                                      ⇐
            }                                                         ⇐
         }                                                            ⇐
      }
   } until BESTCOND is null
   return RULES
}
```

Figure 3.5. The Recovery algorithm for covering one class

current best condition, OTHERGOODCONDITIONS could contain them all and more than MAXSTAR rules could be added.

Note that the `LaplaceAccuracy` used for this comparison is that computed by evaluating the rule over the *entire training set* as opposed to the set of examples over which the rule was created. This will likely be a different number than got the rule selected as either BESTCOND or a member of OTHERGOODCONDS but is necessary to ensure that the rules are applicable to new instances. As mentioned in Section 3.1.3, recalculating `LaplaceAccuracy` over the entire training set gives a better feel for the true predictivity of the rule.

Candidate rules are added to the list of rules to be returned if they are not duplicates of rules already on the list or are not specializations of other rules on the list. If the new redundant rule is more general than another redundant rule already on the rule list, then the previous redundant rule is removed. Also, when a later covering rule (a nonredundant rule) is added that is a duplicate of a redundant rule or generalizes an existing redundant rule, then the redundant copy of the rule is dropped and not counted in the number of

redundant rules generated. Note that adding a more specific rule after a more general rule is allowed but not vice versa. The latter case is detected as a specialization and the rule is not added.

Note that redundant rules are only added during the first iteration of `Recovery-ForOneClass` for each CLASS. During this iteration, all examples belonging to CLASS are used to evaluate the conditions so that the quality of the redundant rules is more likely to be high. Redundant rules could be added at later iterations, but they would be based on fewer examples because some examples are removed during each iteration. We conducted experiments where redundant rules were added at any iteration so long as the redundant rule was within BESTCOND_SIMILARITY of the best rule *for the entire class*. By comparing to the best rule for the class, the quality of the redundant rules was ensured. This was necessary because the covering algorithm will iterate trying to cover all examples of the class. At some point, there may be only a few uncovered examples remaining and the algorithm may have to select a low frequency or inaccurate rule to cover them. Results using this approach were mixed. The accuracy on some data sets improved dramatically but it suffered significantly on others. The best overall results were achieved when only redundant rules generated during the first iteration were added, presumably because these rules have the greatest coverage, therefore this approach was retained in the final system. This issue, however, definitely warrants further investigation.

Note that the examples covered by the redundant rules are not removed. This ensures that the redundant rules added are truly redundant. Removing these redundant rules would leave exactly the set of rules created by the nonredundant version of the algorithm.

The conditions used to create the redundant rules added by `RecoveryForOneClass` are accumulated by `RecoveryFindBestConditions` shown in Figure 3.6. This procedure works the same as in CN2, performing a beam search for the best condition that can be found, but also keeps a set of other "very good conditions" in OTHERGOODCONDS. These are conditions that either were BESTCOND at one time, or were as good as BESTCOND when they were tested but were not retained because they could never beat BESTCOND.

Example of Redundant Rule Generation

This section illustrates the process of adding redundant rules with an example from the vote data set. This data set lists the voting records of members of the U.S. House of

```
procedure RecoveryFindBestConditions( EXAMPLES ) {                    ⇐
   let STAR initially contain only the most general condition ("true")
   let NEWSTAR = {}
   let BESTCOND = null
   let OTHERGOODCONDS = {}                                            ⇐
   while STAR is not empty {
      for each condition COND in STAR {
         for each possible attribute test not already in COND {
            let COND1 = COND & TEST
            evaluate COND1 over EXAMPLES using LaplaceAccuracy
            if COND1 is better than or equal to BESTCOND then {      ⇐
               add COND1 to OTHERGOODCONDS                           ⇐
            }                                                        ⇐
            if COND1 is better than BESTCOND
               then let BESTCOND = COND1
            add COND1 to NEWSTAR
            if size of NEWSTAR > MAXSTAR (a user-defined constant)
               then remove the worst condition in NEWSTAR
         }
      }
      let STAR = NEWSTAR
   }
   return BESTCOND and OTHERGOODCONDS                                ⇐
}
```

Figure 3.6. The Recovery algorithm for finding the best conditions

Representatives on 16 issues. The task is to infer political party (Democrat or Republican) based on voting record. The data set is described in more detail in Section 4.1.

In both the nonredundant rule learner CN2 and Recovery the first rule created by `CN2ForOneClass` for the class Democrat would be

```
IF ((physician fee freeze = "n") AND (adoption of budget = "y"))
  THEN class = "democrat"   [151 0]   LaplaceAccuracy = 0.9935
```

When creating redundant rules with Recovery, the same first rule would be added. Then the process of adding redundant rules would begin. The set of OTHERGOOD-CONDS would contain:

```
IF ((el salvador aid = "n") AND (physician fee freeze = "n"))
  THEN class = "democrat"   [130 0]   LaplaceAccuracy = 0.9924
IF ((superfund right to sue = "n") AND (crime = "n"))
  THEN class = "democrat"   [95 0]    LaplaceAccuracy = 0.9897
IF (physician fee freeze = "n")
  THEN class = "democrat"   [167 1]   LaplaceAccuracy = 0.9882
IF (adoption of budget = "y")
  THEN class = "democrat"   [158 17]  LaplaceAccuracy = 0.8983
IF (handicapped infants = "y")
  THEN class = "democrat"   [105 22]  LaplaceAccuracy = 0.8217
```

The numbers in square brackets show the total number of training examples covered by each rule. The `LaplaceAccuracy` of each is also listed. If BESTCOND_SIMILARITY is set to 0.925, each rule would be compared to the one created from BESTCOND and the first three would be added because they are within 92.5% of the `LaplaceAccuracy` calculated for the best rule. The other two rules fall below the threshold (0.9935 * 0.925 = 0.9190) and would not be added.

The complete rule lists for this data set, both with and without redundancy, are shown in Section 4.4.

Missing Values

During generation, rules are evaluated using the `LaplaceAccuracy` metric, the same as in CN2. The only difference is how the examples containing missing values are handled. When evaluating rules in `RecoveryFindBestConditions`, Recovery uses `LaplaceAccuracy`, as did CN2, but it is calculated slightly differently. As shown in the description of `LaplaceAccuracy` in section 3.1.1, CN2 used the total number of examples being considered as n_{tot}. Recovery considers only the number of examples *with a value for the given attributes*. If all examples have values for all of the attributes in the condition of the rule, then these numbers are the same, but if some examples are missing values for some of the attributes, then the number used for n_{tot} in Recovery will be less than that used in CN2.

It is useful to draw a comparison with taking a survey. Suppose 100 surveys are returned but only 80 people answer the question about eye color and 50 of them have blue eyes. Is it fair to say that 50% of people have blue eyes? Or is it more correct to say that 50/80 = 62.5% of the people have blue eyes? Chapter 4 will investigate this empirically.

The way Recovery handles missing values is significant because many systems, includ-

ing CN2 and C4.5, replace missing values. Some systems simply substitute the most common value for the attribute in place of the missing value while other systems use a probabilistic scheme. If better results are obtained by ignoring the missing values both during training and testing then the approaches used by these other systems should be reconsidered. Results in Section 4.3.3 demonstrate that the handling of missing values can have a substantial impact on the performance of the concept classifier that is learned.

3.2.2 Classification with Recovery

CN2 classifies new instances by searching the final rule list for all matching rules, then letting them vote based on the distribution of examples covered. This was illustrated with an example in Section 3.1.3. This process works well in many cases, but can fail when one matching rule is highly applicable but lacks precision. In this case, the more accurate but less applicable rule may be overshadowed.

Consider the following example of two hypothetical rules and an instance that satisfies both conditions but really belongs to `class = "-"`.

```
IF (some condition) THEN class = "+"      [1000 100]
IF (another condition) THEN class = "-"   [   0 500]
```

Both rules are applicable to the instance being considered. In CN2, the class distributions of these two rules would get summed to [1000 600]. CN2 would then return `class = "+"`. Because the first rule is so frequent, the second rule, which is more accurate, has no chance to win the vote.

Recovery changes the way that the applicable rules vote. As with CN2, Recovery first searches the entire rule set for rules that cover the new instance. Unlike CN2, Recovery does not just sum the class distributions of the rules that fired, but rather, performs a calculation across the class distributions that returns a single value for each rule, then uses this value as a weight. As with CN2, the distribution used during classification is that across all training examples, not just the subset of examples that was being considered in `RecoveryFindBestConditions`. Each rule votes for the class it covers according to this weight. This alleviates the problem of high frequency rules that overshadow more accurate, but less frequent rules. This new metric is called `VectorMagnitude` and is described shortly.

The rules from the example of where CN2 fails during classification are now repeated showing the value of `VectorMagnitude` computed for each class.

```
IF (some condition) THEN class = "+"     [9.1   0.0]
IF (another condition) THEN class = "-"  [0.0 500.0]
```

These values would be summed to get [9.1 500.0] and Recovery would correctly return `class = "-"`.

3.2.3 Rule Weighting Metrics

To compute the weight for a rule, Recovery uses a new metric called `VectorMagnitude`. This metric is similar to that used in the *vector-space* model from information retrieval [13]. It is given by the following formula:

$$\texttt{VectorMagnitude} = (n_c/(n_{tot} - n_c + 1)) * \sqrt{\sum_c (n_c/n_{tot})^2}$$

where

n_c is the number of training examples belonging to class c that are covered by the rule

n_{tot} is the total number of training examples covered by the rule

The square root corresponds to the magnitude of a vector in c-space where c is the number of possible classes. If all examples are aligned to a single axis, the resulting vector has a maximum value. If the examples are distributed across the different classes, the resulting vector will have a lesser magnitude. The idea is to capture the amount of homogeneity among the classes in a single number.

The multiplier before the square root $(n_c/(n_{tot} - n_c + 1))$ accounts for the absolute number of examples covered. It provides a weight corresponding to the number of examples the rule classifies correctly. If all the examples belong to the same class (the rule is perfect), this evaluates to the number of examples covered. If the rule is less than perfect, the weight decreases.

Returning to the example of computing `LaplaceAccuracy` in section 3.1.1, the rule that partitions the data [10 9 1] returns a `VectorMagnitude` of 0.6132. The rule that partitions the data [10 5 5] returns a `VectorMagnitude` of 0.5567. Thus the first rule would be chosen. This is preferable since it is more likely that a later rule can isolate the nine examples in $class_2$ as opposed to dealing with an even split across $class_2$ and $class_3$.

3.3 Summary

This chapter has presented the redundant rule learning system Recovery. It has presented the contributions made by this thesis including:

- the addition of redundancy to covering algorithms
- a better way to handle missing values, namely by ignoring them
- a new metric for rule weighting called vector magnitude

The combination of these changes to a basic covering algorithm result in the redundant rule learning system called Recovery.

CHAPTER 4

RESULTS

This chapter presents the experimental results obtained with Recovery. We first describe the 10 data sets tested against and why each was chosen. Next we describe the other systems that were evaluated, including CN2 [3], RIPPER [4], C4.5 [12], and C4.5rules [12]. Then the overall results are presented. For each of the three contributions made by Recovery (ignoring missing values, weighting rules with vector magnitude, and adding redundant rules) we show the effect of the contribution on the final system.

4.1 The Data Sets

Recovery was evaluated across ten different data sets. The data sets were chosen to provide a variety of different learning tasks to show that Recovery was not specific to any one task. The size of the data set, both in terms of the number of attribute-value combinations and the number of examples, was also important for selection so as to test the scalability of Recovery.

For simplicity, Recovery is programmed only to handle nominal values. These are values that take on a single value from a set of discrete values. This restricted the number of data sets available. To provide more data, one of the data sets was modified by removing the attributes that were continuously valued. This data set is marked with an asterisk ('*') as a reminder that it is modified from its original form. These modifications are mentioned in the descriptions below. Recovery could work on data sets with continuous values if code were added to discretize the values. This is the approach taken by C4.5.

In cases where there was not a testing set of examples available (all data existed in a single set), the set of examples was randomly split into a training set and a testing set. The approximate percentages allocated to training and testing are listed in the descriptions of the data sets below. The partitions were checked to ensure that the class distributions of the examples in the partitions was similar to that of the original set.

All of the data sets used are available from the University of California, Irvine (UCI) repository of machine learning databases [10] with the exception of the synthetic data set that was created for this thesis. The data sets are summarized in Table 4.1. A brief description of each data set and why it was chosen follows.

monk{1,2,3}: The monk's problems are a collection of synthetic classification tasks. There are six attributes, each with two, three, or four possible values. There are a total of 432 combinations of these attributes and possible values and the complete set of combinations provides the test set used by all three problems. monk1 represents the concept (attribute_1 = attribute_2) or (attribute_5 = 1). monk2 represents (attribute_n = 1) for EXACTLY TWO choices of n (in 1,2,...,6). monk3 represents (attribute_5 = 3 and attribute_4 = 1) or (attribute_5 != 4 and attribute_2 != 3) and contains five percent additional noise (instances that are misclassified). These completely specified data sets were used to confirm that the nonredundant covering algorithm forming the core of Recovery achieves results similar to those achieved by CN2.

vote: This data set lists the congressional voting records of members of the U.S. House of Representatives of the 98th Congress on sixteen issues. The task is to infer political party (Democrat or Republican) based on voting record. All records were completely specified. This data set provides another simple test in a domain that has more attributes and hence is somewhat more complex to search.

Table 4.1. Summary of the data sets

	# attributes	*# classes*	*# training examples*	*# test examples*	*missing values?*
monk1	6	2	124	432	no
monk2	6	2	169	432	no
monk3	6	2	122	432	no
vote	16	2	300	135	no
tic-tac-toe	9	2	600	358	no
nursery	8	5	8000	4960	no
synthetic	2	2	120	13	yes
audio	62	24	133	66	yes
soybean	35	19	450	233	yes
anneal*	32	6	798	100	yes

tic-tac-toe: This database encodes the complete set of possible board configurations at the end of tic-tac-toe games where "x" is assumed to have played first. The target concept is "win for x" (i.e., true when "x" has one of 8 possible ways to create a "three-in-a-row"). The original data set consisted 958 examples that were randomly split 63% for training and the remainder for testing. The data are completely specified. This is another large data set to test performance on nontrivial learning tasks.

nursery: The nursery database was derived from a hierarchical decision model originally developed to rank applications for nursery schools. It was used during several years in the 1980s when there was excessive enrollment to these schools in Ljubljana, Slovenia. Rejected applications frequently needed an objective explanation. The original data set was randomly split into a training set that contains approximately 62% of the data with the remainder allocated to testing. There were no missing values in the data. This is the largest data set used and provides a test of scalability.

synthetic: This synthetic data set was created for this thesis. The examples were listed in Figure 1.7. This data set was constructed to include cases where ordinary covering algorithms would generate a less than optimal set of rules leaving room for redundancy to improve the results. Each of the 12 training examples has been repeated 10 times to avoid sparse data problems. The test data includes missing values and previously unseen values and was designed to evaluate the contribution of redundancy.

audio: The original data set consisted of 199 examples that were randomly split 67% for training and 33% for testing. This relatively small data set was used to test the effect of ignoring missing values.

soybean: This is Michalski's famous database of soybean diseases. The original data set was randomly split into a training set that contained 65% of the data and a test set that contained the remaining 35%. The data contain missing values. This mid-sized data set provides a test of how well Recovery's strategy of ignoring missing values works.

anneal*: This data set was modified to remove the six continuously valued attributes,

leaving 32 nominally valued attributes. The asterisk after the name indicates that it is modified from its original form. Many of the examples are incompletely specified. This is a slightly larger test of how well Recovery handles missing values.

4.2 The Algorithms

Recovery was evaluated against a number of popular learning systems to test its performance. The systems and the settings used for each are now briefly described. More information about the algorithms can be found in the background section (Chapter 2).

CN2: This is a popular covering algorithm and one that achieves good results on a wide variety of data sets. Because CN2 is the bases for Recovery, the results from Recovery were tested against those from CN2. The default settings for CN2, including a beam width (maxstar) of 5 and the option to produce unordered rules, were used.

RIPPER: This is another covering algorithm that is popular for text classification tasks. Testing against RIPPER provided a comparison to another covering algorithm. Again, the default settings were used.

C4.5: This is Quinlan's decision tree algorithm and is one of the most popular methods for creating concept classifiers. Comparing against C4.5 provided a test of Recovery's performance against a noncovering algorithm approach to learning. Release 8 of C4.5 was used at the default settings. Results are shown for the pruned versions of the decision trees.

C4.5rules: C4.5rules was used to convert the decision trees produced by C4.5 into production rules. Testing against C4.5rules provides a test against another rule list algorithm, but one that creates those rule lists by a means different than a covering algorithm. Again, the default settings were used.

Recovery: This is the redundant rule learning system introduced in this thesis. Several versions of Recovery were tested and are described as they are presented. For all variants, a beam width of 5 was used and unordered rule sets were produced. When using redundancy, BESTCOND_SIMILARITY was set to 0.925.

4.3 The Results

This section compares the results obtained with Recovery and the other machine learning systems that were tested. First the overall results are given. Then the results are broken down to isolate the effect of each contribution made by Recovery.

When evaluating these results, keep in mind that it is unlikely that any single algorithm will always provide the best results on every data set. The key is to consider selecting an algorithm to use for a particular learning task. A good algorithm is one that consistently performs well. There may be a few cases where it gets beaten by other algorithms, but it provides the best results *overall*.

Results are expressed as accuracies. Accuracy is the percentage of examples the system classifies correctly. It is calculated by dividing the number of examples classified correctly by the total number of examples classified. If 100 examples are classified and 85 are classified correctly, then the accuracy is 85%.

4.3.1 Overall Results

Table 4.2 shows how Recovery fared against each of the other algorithms evaluated. The version of Recovery used was one that creates redundant rules with a BEST-COND_SIMILARITY of 0.925 and uses `VectorMagnitude` to weight the rules during classification.

The columns next to the accuracies show how the other system did relative to Recovery. A '–' means the algorithm performed worse than Recovery. An 'o' means the two systems obtained the same performance. Columns marked with a '+' show cases where the other algorithm performed better than Recovery.

Recovery did at least as well as CN2 on all but one of the data sets tested. On 7 of the 10 data sets, Recovery achieved higher accuracy than CN2. On several of these data sets, (synthetic, audio, soybean, and anneal*) the improvements were substantial. The difference of 13% on anneal* is especially encouraging. It is interesting to note that all the cases of major improvement occur on data sets that contained missing values. These results demonstrate that Recovery is a viable alternative to CN2.

Recovery also fared well versus RIPPER. Recovery outperformed RIPPER on every one of the data sets with the exception of monk3. Some of the improvements obtained with Recovery are substantial. The differences of 22% on monk1, 12% on monk2, and 9.2% on tic-tac-toe are impressive. Clearly Recovery found better rules than RIPPER on most of the data sets tested.

Table 4.2. Results of comparing Recovery to the other algorithms tested

	Recovery	*CN2*	*RIPPER*	*C4.5*	*C4.5rules*
monk1	98.6%	98.6% (o)	76.6% (–)	75.7% (–)	100.0% (+)
monk2	77.3%	75.7% (–)	65.3% (–)	65.0% (–)	66.2% (–)
monk3	90.0%	90.7% (+)	92.6% (+)	97.2% (+)	96.3% (+)
vote	96.3%	95.6% (–)	94.8% (–)	97.0% (+)	94.8% (–)
tic-tac-toe	98.0%	98.0% (o)	88.8% (–)	86.0% (–)	97.8% (–)
nursery	98.3%	97.8% (–)	97.5% (–)	96.6% (–)	97.9% (–)
synthetic	100.0%	92.3% (–)	84.6% (–)	84.6% (–)	84.6% (–)
audio	74.2%	63.6% (–)	71.2% (–)	74.2% (o)	72.7% (–)
soybean	94.4%	88.4% (–)	87.6% (–)	87.1% (–)	90.6% (–)
anneal*	96.0%	83.0% (–)	88.0% (–)	80.0% (–)	87.0% (–)

Versus the decision tree algorithm C4.5, Recovery performed strictly better on seven of the ten data sets. On two, Recovery did somewhat worse indicating that C4.5 found better conditions in those cases, but Recovery posted huge wins on many of the data sets including monk1, monk2, synthetic, and anneal*. The difference of 22.9% on monk1 is especially encouraging. Clearly, the improvements greatly outweigh the losses.

The results of C4.5rules are similar to those with C4.5. Recovery performed better the C4.5rules on eight of the ten data sets. Again, the wins, especially the 16.8% improvement on monk1, seem to largely outweigh the losses.

Overall, these results are very encouraging for Recovery. The new redundant system consistently outperforms the other algorithms on a majority of the data sets. While it may not always produce the best results, it is the best choice overall. Recovery would be a good choice when selecting a system for building a concept classifier.

As mentioned in Chapter 3, there are a number of things that Recovery does differently than CN2. Specifically, Recovery uses a different scheme for handling missing values, a different weighting scheme for classification, and redundancy. The next sections further analyze the data to determine the contribution of each of these items.

4.3.2 The Effect of Ignoring Missing Values

One difference between Recovery and CN2 is the way Recovery handles missing values. CN2 substitutes the most common value for an attribute in cases when an instance has an unknown value for an attribute. Recovery simply ignores the missing values.

To evaluate the effect of ignoring missing values, a version of Recovery that did not

produce redundant rules and that used the CN2 voting scheme was used. This variant of Recovery closely approximates CN2 and should produce the same results on all data sets that do not contain missing values. CN2 handles missing values by substituting the most frequent value and then discounting the confidence associated with that value. This weighting was not implemented in Recovery and thus results on data sets that contain missing values differ. Table 4.3 shows the effects of the different methods for handling missing values.

In cases where there were no missing values in the data (monk1, monk2, monk3, vote, nursery, tic-tac-toe), results with Recovery are nearly identical to those of CN2. The minor differences of 0.2% on monk3 and 0.3% on nursery can be attributed to how rules that evaluate to exactly the same value are ordered during the beam search.

In cases where there are missing values in the data (synthetic, soybean, anneal*, audio), Recovery did substantially better in all but one case. The poor performance on the synthetic data set is misleading because it is due to a single misclassified test instance.

These results are somewhat surprising but important. Intuitively, it seems like a good idea to substitute the most frequent value for an attribute in cases of missing values, but results suggest that it is better to ignore the missing values when computing the metric. The fact that some algorithms (namely decision trees) *must* have values for each attribute appears to be a problem. Considering how simple it is to ignore the missing values in computing rule weights, this technique appears to be a huge win.

Table 4.3. Results showing the effect of ignoring missing values

	Recovery *(without redundancy; CN2 weighting)*	*CN2*	missing values?
monk1	98.6%	98.6% (o)	no
monk2	75.7%	75.7% (o)	no
monk3	90.5%	90.7% (+)	no
vote	95.6%	95.6% (o)	no
tic-tac-toe	98.0%	98.0% (o)	no
nursery	98.1%	97.8% (–)	no
synthetic	84.6%	92.3% (+)	yes
audio	66.7%	63.6% (–)	yes
soybean	93.1%	88.4% (–)	yes
anneal*	94.0%	83.0% (–)	yes

4.3.3 The Effect of Vector Magnitude

The next effect to examine is that produced by changing the way rules are weighted when they are used to classify new instances. As discussed in the previous section, CN2 collects all rules that cover the new instance and sums the class distributions of those rules to determine the class. Recovery uses a different scheme where rules are weighted by the vector magnitude of the training examples they cover.

Results from two variants of Recovery are shown in Table 4.4. In both variants, redundancy has been disabled to isolate the contribution of the vector magnitude weighting. The first variant of Recovery, shown on the left, uses the vector magnitude weighting. The other variant of Recovery uses the same weighting scheme as CN2. Results from CN2 are also listed for comparison. The differences between Recovery with CN2 weighting and CN2 can be attributed to the fact that Recovery ignores missing values.

In all cases except with monk3, using vector magnitude does at least as well as using the CN2 weighting scheme. In several cases (nursery, audio, soybean, and anneal*) vector magnitude does better.

These results make intuitive sense because the CN2 weighting scheme gives preference to "heavy-hitters" – rules that may not be very accurate but cover many examples. If just one of these rules applies to a new instance, then it would be difficult for less applicable, but more precise rules, to influence the vote. Vector magnitude accounts for this, leveling the playing field somewhat.

Table 4.4. Results showing the effect of using vector magnitude

	Recovery *(without redundancy; VMAG weighting)*	**Recovery** *(without redundancy; CN2 weighting)*		*CN2*	
monk1	98.6%	98.6%	(o)	98.6%	(o)
monk2	75.5%	75.7%	(+)	75.7%	(+)
monk3	90.5%	90.5%	(o)	90.7%	(+)
vote	95.6%	95.6%	(o)	95.6%	(o)
tic-tac-toe	98.0%	98.0%	(o)	98.0%	(o)
nursery	98.3%	98.1%	(–)	97.8%	(–)
synthetic	84.6%	84.6%	(o)	92.3%	(+)
audio	69.7%	66.7%	(–)	63.6%	(–)
soybean	93.6%	93.1%	(–)	88.4%	(–)
anneal*	96.0%	94.0%	(–)	83.0%	(–)

Using vector magnitude rarely hurts. It appears to help enough in several cases to be worthy of consideration as a rule weighting metric. The benefits of using vector magnitude will become more apparent when considered in conjunction with redundant rules. These results will be explored shortly.

4.3.4 The Effect of Redundancy

Finally, it is appropriate to examine how the results change when redundancy is added. Two variants of Recovery are compared. The first variant includes redundancy and the second does not. In Table 4.5, the left column shows the results when redundancy is used. The right column shows the results without redundant rules. Both variants used the vector magnitude metric for weighting rules during classification. These results isolate the contribution of redundancy.

In general, the addition of redundancy improves the results yet again. On five of the data sets tested (monk2, vote, synthetic, audio, and soybean), there was a noticeable improvement when redundant rules were added. The gains are not tremendous, but they show a consistent win for redundancy. On another four sets (monk1, tic-tac-toe, nursery, and anneal*), the scores with redundant rules were no different than without. In these cases, redundancy is not hurting. Only on monk3 was there a decrease in performance with when redundancy was added and this decrease was small. Overall it appears that the addition of redundant rules to the learner helps.

Table 4.5. Results showing the effect of redundancy

	Recovery *(with redundancy; VMAG weighting)*	**Recovery** *(with redundancy; CN2 weighting)*	
monk1	98.6%	98.6%	(o)
monk2	77.3%	75.5%	(–)
monk3	90.0%	90.5%	(+)
vote	96.3%	95.6%	(–)
tic-tac-toe	98.0%	98.0%	(o)
nursery	98.3%	98.3%	(o)
synthetic	100.0%	84.6%	(–)
audio	74.2%	69.7%	(–)
soybean	94.4%	93.6%	(–)
anneal*	96.0%	96.0%	(o)

The previous results were run using `VectorMagnitude` to weight the rules during classification. We also ran Recovery using redundancy but retaining the CN2 weighting scheme to see if there is a combined effect between redundant rule generation and `VectorMagnitude`. The results when the CN2 voting scheme was used are significantly worse. On five of the data sets, using `VectorMagnitude` greatly improved the results when redundancy was used. These results are shown in Table 4.6.

These results show that there is a synergistic effect between redundancy and `VectorMagnitude`. This can be explained by considering that redundancy will likely add more rules to the system. This could compound the "heavy-hitter" problem whereby high frequency rules overshadow less frequent but more accurate rules. Adding more low frequency rules may have no effect if there are a few high frequency rules that dominate. Using `VectorMagnitude` decreases the contribution of frequency in favor of accuracy. The combination of `VectorMagnitude` and redundant rules help build better classifiers.

4.4 Analysis of Redundant Rules

Redundancy is a feature of Recovery that can be enabled or disabled. Recovery will always generate the same set of nonredundant rules. This allows the set of purely redundant rules to be analyzed for their contribution. This section lists the rule sets created for the vote data set described in Section 4.1. Figure 4.1 shows the 20 nonredundant rules created by Recovery.

Table 4.6. Results showing the combined effect

	Recovery *(with redundancy; VMAG weighting)*	**Recovery** *(without redundancy; VMAG weighting)*	
monk1	98.6%	97.2%	(+)
monk2	77.3%	77.6%	(–)
monk3	90.0%	90.0%	(o)
vote	96.3%	96.3%	(o)
tic-tac-toe	98.0%	93.9%	(+)
nursery	98.3%	98.1%	(+)
synthetic	100.0%	100.0%	(o)
audio	74.2%	69.7%	(+)
soybean	94.4%	94.4%	(o)
anneal*	96.0%	92.0%	(+)

```
IF (physician fee freeze=n AND adoption of budget=y)
  THEN class=democrat [151 0],
IF (synfuels cutback=y AND physician fee freeze=n)
  THEN class=democrat [79 0],
IF (el salvador aid=n AND physician fee freeze=n)
  THEN class=democrat [130 0],
IF (physician fee freeze=u AND water project=y)
  THEN class=democrat [4 0],
IF (physician fee freeze=n AND education spending=y)
  THEN class=democrat [24 0],
IF (el salvador aid=n AND synfuels cutback=y AND adoption of budget=n)
  THEN class=democrat [5 0],
IF (adoption of budget=u AND crime=y AND religious groups in schools=y)
  THEN class=democrat [2 0],
IF (religious groups in schools=u AND mx missile=y)
  THEN class=democrat [4 0],
IF (duty free exports=y AND export act=u AND anti satellite test ban=n)
  THEN class=democrat [3 0],
IF (physician fee freeze=y AND synfuels cutback=n AND
    adoption of budget=n AND superfund right to sue=y)
  THEN class=republican [0 72],
IF (physician fee freeze=y AND export act=y AND immigration=y)
  THEN class=republican [0 40],
IF (physician fee freeze=y AND education spending=y AND
    immigration=y AND adoption of budget=n)
  THEN class=republican [0 42],
IF (physician fee freeze=y AND aid to nicaraguan contras=u)
  THEN class=republican [0 7],
IF (physician fee freeze=y AND export act=y AND superfund right to sue=y)
  THEN class=republican [0 54],
IF (physician fee freeze=u AND mx missile=u)
  THEN class=republican [0 2],
IF (physician fee freeze=y AND duty free exports=u)
  THEN class=republican [0 7],
IF (physician fee freeze=y AND handicapped infants=y AND water project=n)
  THEN class=republican [0 5],
IF (handicapped infants=u AND anti satellite test ban=n)
  THEN class=republican [0 1],
IF (physician fee freeze=y AND crime=u)
  THEN class=republican [0 6],
IF (export act=n AND water project=y AND handicapped infants=n AND
    education spending=y AND superfund right to sue=y)
  THEN class=republican [0 11]
```

Figure 4.1. Nonredundant rules produced by Recovery for the vote data set.

Figure 4.2 shows the additional nine redundant rules that would be added by Recovery with redundancy enabled and a BESTCOND_SIMILARITY of 0.925. Note that in Section 3.2.1 three redundant rules were listed for the class Democrat. This was indeed the case, but, as mentioned, when a later covering (nonredundant) rule is added that is a duplicate of a redundant rule or is more general that an existing redundant rule, then the redundant rule is dropped. That was indeed the case here.

Notice that many of these rules are of high frequency and are also highly accurate. None of the rules covered fewer that 30 of the 300 training examples and none of the redundant rules misclassifies more than one training example. These are high quality rules that have the potential to improve classification performance.

Note that the second redundant rule listed is a more general version of the first nonredundant rule. The more general rule is applicable to several more cases and is

```
IF (superfund right to sue=n AND crime=n)
  THEN class=democrat [95 0]
IF (physician fee freeze=n)
  THEN class=democrat [167 1]

IF (physician fee freeze=y AND synfuels cutback=n AND
    adoption of budget=n AND duty free exports=n)
  THEN class=republican [0 72]
IF (physician fee freeze=y AND synfuels cutback=n AND
    adoption of budget=n AND education spending=y)
  THEN class=republican [0 69]
IF (physician fee freeze=y AND immigration=y AND
    synfuels cutback=n)
  THEN class=republican [0 51]
IF (education spending=y AND physician fee freeze=y AND
    export act=y)
  THEN class=republican [0 49]
IF (education spending=y AND physician fee freeze=y AND
    synfuels cutback=n)
  THEN class=republican [1 76]
IF (physician fee freeze=y AND water project=n AND
    export act=y)
  THEN class=republican [0 31]
IF (physician fee freeze=y AND water project=n AND
    immigration=y)
  THEN class=republican [0 30]
```

Figure 4.2. Redundant rules produced by Recovery for the vote data set.

only slightly less accurate. Keeping this rule might help correctly classify cases where the value for the attribute `adoption of budget` is missing.

4.5 Number of Redundant Rules

It is interesting to consider the number of redundant rules created by Recovery. Table 4.7 shows the number of rules created without redundancy and the number of redundant rules added by Recovery. In many cases, very few redundant rules are produced. This is caused by using fairly strict criterion for adding redundant rules. The limitation of adding redundant rules only during the first iteration per class and the limits imposed by setting BESTCOND_SIMILARITY to 0.925 mean that the redundant rules retained are high quality. Either of these constraints can be relaxed, but at the risk of introducing unreliable rules into the system. Experiments were also run where redundant rules were allowed to be added at later iterations and with different values for the threshold (0.90 and 0.95) and the current configuration gave the best overall results.

4.6 Summary

Experimental results show that using Recovery is a win because it consistently outperforms a number of other widely used learning systems. The improved results come from a variety of changes made to the basic covering algorithm. The biggest improvement came from ignoring missing values when creating rules and using them for classification. The new metric `VectorMagnitude` also showed improvement over the simple voting scheme used by CN2. Improvements were also seen when redundant rules were added to the

Table 4.7. Summary of the number of rules produced

	nonredundant rules	*redundant rules*
monk1	23	1
monk2	65	3
monk3	23	2
vote	20	9
tic-tac-toe	29	6
nursery	298	0
synthetic	2	2
audio	26	86
soybean	36	70
anneal*	35	6

system. Any of these contributions in isolation provide improvement over existing systems but the combination of the three make Recovery a good choice when selecting a system for building concept classifiers.

CHAPTER 5

CONCLUSIONS

Chapter 1 of this thesis introduced the idea of redundancy and explained how machine learning could benefit from it. The following claim was made:

Exploiting redundancy can improve the predictive accuracy of conceptual classifiers.

To substantiate the claim, a redundant rule learning system called Recovery was developed. Chapter 3 presented this algorithm. Recovery works much like the ordinary covering algorithm CN2, but also learns redundant rules that can be used to classify new instances. Recovery also ignores missing values both during rule generation and classification and uses a new rule evaluation metric called vector magnitude. All three of these modifications contribute to its success.

Recovery was tested against a number of other systems including CN2, RIPPER, C4.5, and C4.5rules. Results shown in Chapter 4 demonstrate that the classifiers built by Recovery provide an overall increase in predictive accuracy versus the classifiers built by the other systems. Given the consistent improvements, Recovery appears to be a viable alternative to other popular learning systems and is worthy of consideration when selecting a system to build classifiers.

5.1 Contributions

Recovery makes three contributions to the field of machine learning. First, it makes use of redundant rules. Results showed that the use of redundancy improves classification performance. Chapter 3 showed how Recovery accumulates redundant rules and how it evaluates them for inclusion in the final rule list. The redundant rules generated by Recovery add robustness to the classifiers that are created. The gains achieved by adding redundancy were small, but consistent.

Another contribution Recovery makes is the `VectorMagnitude` metric. This new metric is intuitive and considers not only the accuracy of a rule, but also the distribution of examples that the rule misclassifies. This is helpful for alleviating the problem of high frequency rules overshadowing less frequent but more accurate rules ("heavy-hitters"). Recovery uses the vector magnitude metric to weight rules during classification. Results show that it outperforms the weighting scheme used by CN2 and when combined with redundancy provides an even greater benefit.

Finally, Recovery handles missing values by ignoring them during both rule generation and new instance classification. This is different than most machine learning systems, many of which substitute the most frequent value in place of the missing value. Results showed that when missing values were ignored the classifiers built achieved higher accuracy than when values were substituted. The only other learning system that takes this approach is RIPPER when it is treating attributes as set-valued attributes. This approach merits use in other learning systems because of the significant improvements observed when this technique was used.

5.2 Applications

Recovery is a general purpose system for creating concept classifiers. It can be used on any classification problem where other systems such as CN2, RIPPER, and C4.5 can be used. Although Recovery currently cannot handle continuously valued attributes, this could be easily remedied by adding code that discretizes those values. As long as the problem can be suitably represented as examples described by attribute-value pairs, Recovery can be used to create a classifier.

Other learning systems have been used on a wide variety of classification tasks ranging from medical diagnosis, to credit evaluation, to text classifiers. Recovery seems especially well suited to tasks of text classification. In text classification, the attributes for the learning system are usually the words in the vocabulary. Each document is described by the words it contains. Because the words in a document are relatively sparse with respect to the words in the system's vocabulary, many "features" will have missing values. Ignoring the missing values was a huge win for Recovery and these improvements should carry over to text classification tasks. Also, since some words are very frequent, while other are not so frequent, there is the problem of "heavy-hitter" rules. Using vector magnitude can help alleviate this problem and should help in text classification. Also,

the redundant rules created by Recovery could help with the problems of synonymy that abound in text classification tasks.

5.3 Significance and Efficiency

The results achieved with Recovery are especially encouraging considering the simplicity of the techniques used. The algorithm essentially requires no extra effort to accumulate the redundant rules and minimal effort to check them against the threshold and add them to the rule list. This means Recovery has the same computational complexity as CN2 except for the extra effort required to check the redundant rules for duplication and subsumption. Recovery, therefore, is scalable both in the number of training examples in the data and the number of attributes and possible values that these data contain.

The approach used in Recovery is simpler than the approach used by DAIRY. DAIRY made two passes through the training data. The first pass performed "recycling" whereby covered examples were retained at a discounted weight. The second pass was a search for completely redundant rules. Recovery avoids the need for a weighting scheme and creates the redundant rules in a single pass.

Recovery also eliminates the artificial limit of two attribute-value pairs per redundant rule. This limit was used in both DAIRY and the early redundant learning system by Cestnik and Bratko. Both these systems imposed the limit to reduce the search space that must be explored. It is indeed necessary to limit the search but Recovery does this through the beam search rather than by limiting the number of attribute-value pairs that a rule can contain. Work with Recovery shows that a limit is not necessary and rules learned by Recovery can contain any number of attribute-value pairs.

Recovery is an integrated approach that finds redundant rules during the normal covering algorithm search and adds these to the rule list. Even this simple approach has shown improvement over existing algorithms for creating concept classifiers.

5.4 Future work

More complicated schemes for generating redundant rules were tried but the problem of combinatorial explosion was encountered. It is desirable to explore as much of the search space as possible, but this means exploring huge numbers of attribute-value combinations. Steps must be taken to prune the search space. This can be done through limits on the number of attribute-value pairs allowed in a rule or limits on the total number of redundant rules that can be created. In Recovery, the beam search limits the exploration

but there may be ways to expand the search while avoiding the problems with a near exhaustive search.

Another idea for future work would be to experiment with the number of training examples required to build a system. It is possible that the addition of redundancy reduces the number of training examples required to build a classifier. One way to test this would be to systematically reduce the number of training examples presented to the learning systems and track the classification performance as the size of the training set decreases.

This thesis introduced `VectorMagnitude` as a new metric for weighting rules during classification. This method has potential for use during rule generation. Experiments were run using it instead of `LaplaceAccuracy` to evaluate rules during generation but the weighting term used is too biased toward rules that cover many examples. More experimentation is necessary, but we feel that vector magnitude could also be a useful metric for evaluating rules as they are generated.

As a proof of concept, Recovery shows that redundancy can improve machine learning. Results substantiate the claim that exploiting redundancy can improve the accuracy of concept classifiers. Even though the techniques used in this work are simple, they provide a starting point for further exploration of ways that redundancy can add robustness to machine learning.

REFERENCES

[1] Cestnik, B., and Bratko, I. Learning redundant rules in noisy domains. In *ECAI'88: proceedings of the 8th European Conference on Artificial Intelligence* (1988), B. Radig, Ed., Pitman Publishing, pp. 348–350.

[2] Clark, P., and Boswell, R. Rule induction with CN2: Some recent improvements. In *Machine Learning – EWSL - 91: proceedings European Working Session on Learning, Porto, Portugal* (1991), Y. Kodratoff, Ed., Springer-Verlag, pp. 151–163.

[3] Clark, P., and Niblett, T. The CN2 induction algorithm. *Machine Learning Journal 3*, 4 (1989), 261–283.

[4] Cohen, W. W. Fast effective rule induction. In *Machine Learning: Proceedings of the Twelfth International Conference* (Tahoe City, California, 1995), A. Prieditis and S. Russell, Eds., Morgan Kaufmann Publishers.

[5] Cohen, W. W. Learning trees and rules with set-valued features. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* (Portland, Oregon, 1996), AAAI Press.

[6] Hsu, D., Etzioni, O., and Soderland, S. A redundant covering algorithm applied to text classification. In *Learning for Text Categorization: Papers from the 1998 Workshop at AAAI* (Menlo Park, California, 1998), M. Sahami, Ed., American Association for Artificial Intelligence, AAAI Press, pp. 18–25. Technical Report WS-98-05.

[7] Michalski, R. S. A theory and methodology of inductive learning. *Artificial Intelligence 20* (1983), 111–161.

[8] Michalski, R. S., Mozetic, I., Hong, J., and Lavrac, N. The multi-purpose incremental learning system aq15 and its testing application to three medical domains. In *Proceedings of the American Association for Artificial Intelligence Conference (AAAI)* (Philadelphia, PA, 1986), pp. 1041–1045.

[9] Mitchell, T. M. *Machine Learning.* McGraw-Hill, 1997.

[10] Murphy, P. M., and Aha, D. W. *UCI Repository of machine learning databases.* Irvine, California: University of California, Department of Information and Computer Science, 1994. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[11] Quinlan, J. R. Induction of decision trees. *Machine Learning 1* (1986), 81–106.

[12] Quinlan, J. R. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[13] SALTON, G. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, MA, 1989.

[14] UTGOFF, P. E. ID5: An incremental ID3. In *Proceedings of the 5th International Conference on Machine Learning* (1988), pp. 107–120.