INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Machine Learning as Massive Search

by

Richard B. Segal

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by ______________________________
(Chairpersonof Supervisory Committee)

Program Authorized
to Offer Degree Computer Science and Engineering

Date July 18, 1997

UMI Number: 9807027

Copyright 1997 by
Segal, Richard B.

All rights reserved.

UMI Microform 9807027
Copyright 1997, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© Copyright 1997
Richard B. Segal

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature Richard B. Segal

Date July 18, 1997

University of Washington

Abstract

Machine Learning as Massive Search

by Richard B. Segal

Chairperson of Supervisory Committee: Associate Professor Oren Etzioni
Department of Computer Science
and Engineering

Machine learning is the inference of general patterns from data. Machine-learning algorithms search large spaces of potential hypotheses for the hypothesis that best fits the data. Since the search space for most induction problems grows exponentially in the number of features used to describe the data, most induction algorithms use greedy search to minimize search cost. Greedy search is a polynomial-time algorithm that achieves its efficiency by exploring only a tiny fraction of all hypotheses. While greedy search has good performance, it often misses the best hypotheses.

This thesis proposes massive search as an alternative to greedy search. Massive search aggressively searches as many hypotheses as possible in the time available. Since massive search explores a larger portion of the hypothesis space, it is less likely to miss good hypotheses. This thesis develops a massive-search algorithm for rule learning called Brute. Experiments with Brute show that massive search is both practical and effective. Brute can completely search the hypothesis spaces of most benchmark problems in only a few minutes. Brute learns better rules than greedy search on 13 of 18 databases, while performing equally well on the remaining five. We demonstrate massive search's wide applicability by extending Brute to handle data-mining and classification problems with comparable results.

TABLE OF CONTENTS

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1 Motivation 1

1.2 Overview 2

1.2.1 Efficiency 3

1.2.2 Effectiveness 5

1.2.3 Applications 6

1.3 Scientific Contributions 7

1.4 Organization 8

Chapter 2: Rule Learning 10

2.1 Problem Definition 10

2.2 Propositional Rule Learning 13

2.3 Existing Solutions 14

2.3.1 Estimating Accuracy and Coverage 17

2.3.2 Idealized Algorithm 20

2.3.3 Real Systems 27

Chapter 3: Brute 29

3.1 Basic Algorithm 29

3.2 Rule Pruning 32
3.2.1 Branch-and-Bound Pruning 35
3.2.2 Subsumption Pruning 38
3.2.3 Dynamic Reorganization 42
3.2.4 Search Algorithm 43
3.2.5 Depth-Bound Pruning 47
3.2.6 Summary 48
3.3 Rule Evaluation 50
3.3.1 Bit-Vectors 52
3.3.2 Partitioning and Sorting 56
3.4 Complexity 61
3.4.1 Equality Tests 61
3.4.2 Inequality Tests 62
3.4.3 Numerical Attributes 63
3.4.4 Effect of Depth Bound 64
3.4.5 Total Complexity 65
3.5 Performance Analysis 66
3.6 Related Work 70
3.7 Summary 72

Chapter 4: Inductive Performance **73**
4.1 Oversearching 74
4.2 Overfitting 78
4.3 Related Work 89
4.4 Summary 91

Chapter 5: **Data Mining** 93
5.1 Boeing Manufacturing Domain 93
5.2 Problem Definition 95
5.3 Existing Solutions 97
5.3.1 Decision Trees 97
5.3.2 Gold-digger 101
5.4 Data Mining with Brute 106
5.4.1 Preventing Redundancy 106
5.4.2 Rule Pruning 114
5.4.3 Empirical Analysis 117
5.5 Related Work 122
5.6 Summary 125

Chapter 6: **Classification** 126
6.1 Brute-greedy 127
6.2 BruteDL 130
6.2.1 Homogeneous Decision Lists 133
6.2.2 Implications 134
6.2.3 Algorithm 135
6.2.4 Experimental Results 137
6.2.5 Critique 142
6.3 Related Work 143
6.4 Conclusion 145

Chapter 7: **Conclusion** 147
7.1 Summary 147
7.1.1 Is massive search feasible? 147

7.1.2 Does massive search learn better rules than greedy search? . . 149
7.1.3 In what range of tasks is massive search appropriate? 151
7.2 Future Work 154
7.2.1 Better Evaluation Functions 154
7.2.2 Scalability 155
7.2.3 Learning Classifiers 156
7.2.4 First-Order Learning 157
Bibliography **160**

LIST OF FIGURES

2.1 High-level description of the rule-learning problem. 11
2.2 Sample test set and goal predicate. 16
2.3 Contour graph of Laplace accuracy. 19
2.4 Greedy algorithm for rule learning. 21

3.1 Brute's search space with duplicates. 30
3.2 Brute's search space with duplicates removed. 31
3.3 Brute's algorithm for rule learning. 33
3.4 Subsumption pruning. 41
3.5 Branch-and-bound pruning with lexical ordering. 44
3.6 Branch-and-bound pruning with dynamic reorganization. 45
3.7 Estimating accuracy using example caching. 51
3.8 Estimating accuracy using bit-vectors. 53

4.1 Quinlan and Cameron-Jones' oversearching experiments. 75
4.2 Comparison of beam-512 search and greedy search. 76
4.3 Inductive performance of layered search. 79
4.4 Comparison of beam-512 search and layered search. 80
4.5 Comparison of Laplace-depth and Laplace accuracy. 85
4.6 Comparison of exhaustive and beam-512 search for Laplace accuracy. 86
4.7 Comparison of exhaustive and beam-512 search for Laplace-depth. . . 87
4.8 Comparison of Laplace-depth and Laplace accuracy for exhaustive search. 88

4.9 Comparison of exhaustive and greedy search using Laplace-depth. . . 89
4.10 Comparison of exhaustive and layered search using Laplace-depth. . . 90

5.1 High-level description of the data-mining problem. 96
5.2 Sample decision tree. 98
5.3 Converting a decision tree to rules. 99
5.4 The failure of decision-tree splitting criteria. 100
5.5 Pseudocode for Gold-digger. 103
5.6 Algorithm for filtering trivial specializations. 109
5.7 Algorithm for generating correlated rules. 113
5.8 Algorithm for filtering rules with similar numeric thresholds. 115
5.9 Effectiveness of Brute's similarity filters. 121

6.1 Generalized algorithm for learning decision lists. 128
6.2 Comparison of Brute-greedy and Greedy-greedy. 129
6.3 Comparison of Brute-greedy and C4.5. 130
6.4 Comparison of BruteDL and Brute-greedy. 138
6.5 Comparison of modified BruteDL and BruteDL. 139
6.6 Comparison of modified BruteDL and Brute-greedy. 140
6.7 Comparison of modified BruteDL and C4.5. 141

LIST OF TABLES

2.1 Sample database language. 15
2.2 Description of benchmark databases. 25
2.3 Greedy search's inability to maximize Laplace accuracy. 26

3.1 Rationality of standard evaluation functions. 37
3.2 The effectiveness of rule pruning. 49
3.3 Typical speedup afforded by bit-vectors. 55
3.4 Comparison of rule-evaluation methods. 60
3.5 Brute's complexity. 66
3.6 Brute's running time. 67
3.7 Description of large databases. 68
3.8 Brute's running time on large databases. 69

4.1 Rules learned using massive and layered search. 81
4.2 Analysis of a high-coverage overfit rule. 82

5.1 Comparison of Gold-digger and C4.5. 105
5.2 Comparison of Brute, Gold-digger, and C4.5. 119
5.3 Effectiveness of similarity pruning. 122

6.1 Running times and search depths for BruteDL. 142

ACKNOWLEDGMENTS

This thesis would not have been possible without the help of many people. My advisor Oren Etzioni has taught me to be a good researcher, provided many insights into my research, and done wonders to improve my writing skills. Many thanks to Steve Hanks and Dan Weld for serving on my thesis committee, making many contributions to this thesis, and providing a stimulating learning environment.

The idea for this thesis arose from a joint project with Patricia Riddle on a Boeing data-mining application. Pat was extremely instrumental in the development of Brute and my understanding of data mining. Pat also deserves thanks for serving on my reading committee and providing many suggestions and comments that have been incorporated into this thesis.

I would not have been able to complete my thesis if it were not for my many friends who helped make my stay in Seattle so special. Thanks go to the SPUDS softball team for many enjoyable games and to the Pastry-Powered T(o)uring Machines cycling group for many great rides and adventures. Thanks to Jim and Christine Ahrens, Joel Beckerman, Andy and Debbie Berman, Jackie Bricker, Lauren Bricker, Ron Critchfield, Jennifer Fisch, Marc Friedman, Keith Golden, Dave Johnson, Neal Lesh, Ruben Ortega, Mary Kaye Rodgers, Mike Salisbury, Erik Selberg, and Ellen Spertus for being good friends and making graduate school memorable.

I thank Jeff Kephart, my manager at IBM, for his patience and understanding while I finished my thesis at IBM. The last few months at IBM have been great, and I look forward to the years to come.

Finally, I wholeheartedly thank my fiancée Joanna Labendz for her incredible love and support. Joanna has been a tremendous help during the writing process. In addition to being a wonderful companion, Joanna has painstakingly read this thesis many times and has made countless improvements to the text. I hope to be as supportive in her endeavors.

Chapter 1

INTRODUCTION

1.1 Motivation

Machine learning is the inference of general patterns from data. Mitchell [1982] characterizes machine learning as a search for the hypothesis that best describes the data from an extremely large set of potential hypotheses. The large number of potential hypotheses has prevented most inductive algorithms from evaluating every hypothesis. Most algorithms use greedy search to find a good rule by evaluating only a tiny fraction of all hypotheses.

While greedy search performs well on many problems, it has limitations. For most learning problems, greedy search cannot guarantee finding the hypothesis that maximizes the algorithm's evaluation metric. As a result, greedy search often misses the best hypotheses.

Greedy search's limitations are acceptable only if more extensive search is impractical. However, computer technology has progressed such that many greedy learning algorithms take only a few minutes to process most databases. In key application areas such as data mining, the large potential benefits of machine learning can justify hours or even days of computation if the extra computation uncovers better patterns.

This thesis explores the use of massive rather than greedy search for machine learning. Massive search aggressively explores as much of the hypothesis space as possible in the time allotted. When given enough time, massive search algorithms can search the entire hypothesis space and return the hypothesis that best describes the data

according to the algorithm's evaluation metric. By exploring a larger percentage of the hypothesis space, massive-search algorithms find hypotheses that better describe the available data.

1.2 Overview

This thesis addresses three key questions about massive search: is massive search feasible? does it provide better solutions than greedy search? and in what range of machine-learning applications is massive search appropriate? Our approach is to investigate massive search in the context of a particular learning task and show how results on this task generalize to other learning problems.

The choice of a learning task is difficult. Concept learning would be a natural choice since it is one of the most important problems in machine learning. Concept learning is the identification of a concept's definition from positive and negative instances of the concept being studied. Solutions to the concept-learning problem must identify the best concept description from a doubly-exponential set of possible concept descriptions. Unfortunately, this doubly-exponential set is currently too large to explore *directly* using massive search.[1] As a result, we focus on a less complex problem for which massive search can be more readily applied.

The *rule-learning problem* is the learning of a single conjunctive rule that best describes a subset of the available data. Rule learning is ideal for investigating massive search for two reasons. First, since the search space for rule learning grows singly exponentially, it is more amenable to massive search but is still challenging enough that most rule-learning algorithms use greedy or beam search to minimize costs [Michalski 69, Pagallo&Haussler 90, Clark&Niblett 89, Provost *et al.* 93]. Second, rule learning is an important subproblem of many other learning tasks. Many classification algorithms including AQ [Michalski 69], Greedy3 [Pagallo&Haussler 90], and

[1] While direct search is prohibitively expensive, Chapter 6 shows that concept-learning algorithms can still benefit from massive search.

CN2 [Clark&Niblett 89] use a rule-learning component in its inner loop. Rule learning is also the primary task in many data-mining applications [Riddle *et al.* 94, Agrawal *et al.* 96, Provost *et al.* 93].

We evaluate massive search by designing and implementing a massive-search algorithm for rule learning. Building a working system allows us to answer empirically questions about the efficiency and effectiveness of massive search. Furthermore, by embedding a working system in other machine-learning applications, we can determine whether the benefits provided by massive search for rule learning apply to other areas of machine learning.

The massive search algorithm we developed is called Brute. Brute was designed to meet our goals of efficiency, effectiveness, and wide applicability. Each of the goals introduce many challenges to Brute's design. The following sections describe the challenges for each goal in detail.

1.2.1 Efficiency

The simplest algorithm for rule learning using massive search is to enumerate all possible rules, evaluate how accurately each rule describes the data, and return the best rule found. The following formula characterizes the execution time for this algorithm:

$$\mathit{ExecutionTime} = \mathit{NumberOfRules} \times \mathit{RuleEvaluationCost}.$$

The size of the hypothesis space for any rule-learning problem grows exponentially with both the number of features used to describe each instance and the number of values allowed for each feature. If there are A features and V values per feature in a given database, then a massive search algorithm must consider 2^{AV} hypotheses to find the best rule.

Since rule evaluation requires verifying the correctness of the current rule against each example in the database, the cost of rule evaluation grows linearly with the

number of training instances. If we let N denote the number of examples in the database and C represent the amount of CPU time required to process each example, then the total execution time for the naive algorithm is

$$ExecutionTime = 2^{AV} \times C \times N.$$

Even with optimistic values for C and N, the cost of enumeration is prohibitively expensive for modest values of A and V. If we set $C = 1\mu s$ and $N = 1,000$ examples, a database described by ten features ($A = 10$) that can take on five values ($V = 5$) would take over 35,000 years to analyze. Brute has to dramatically improve on the naive algorithm if there is any chance of massive search being practical.

Three parameters affect the execution time of the naive algorithm: the number of rules, the number of examples, and the cost of processing each example. Each of these parameters presents an opportunity for improving efficiency.

Brute reduces the number of rules that must be searched by aggressively pruning portions of the search space that can be proven not to contain the best hypothesis. We develop several new pruning techniques that can prove a substantial fraction of the search space is uninteresting and does not have to be evaluated. Rule pruning is so effective that the highest ranking rule can be found from hypothesis spaces as large as 10^{60} rules by evaluating as few as 10^{6} rules.

We consider two techniques for reducing the cost of rule evaluation. The first technique, *example caching*, eliminates the need to evaluate every rule on every example. Example caching organizes the search space in a tree structure such that the examples consistent with each node are a superset of the examples consistent with each of its children. By caching the examples consistent with each parent, each child can be evaluated by analyzing its performance on the subset of examples matched by its parent. The second technique, partitioning, shares the cost of rule evaluation by analyzing groups of similar rules using a single pass through the database.

While code optimization can reduce the execution constant C, the data structures

employed are also important. We present a novel data structure and evaluation technique that reduces the execution constant for rule learning by as much as fifteen times. While this new representation is not compatible with partitioning, it provides a substantial performance gain over partitioning for all but the largest databases.

The combination of these three techniques achieve an efficient algorithm for rule learning. Brute can completely analyze the hypothesis spaces for most benchmark learning problems available at the UCI Machine Learning Data Repository [Murphy 94] and performs admirably on several Boeing manufacturing databases.

1.2.2 Effectiveness

Assuming the computational costs can be managed, the use of massive search appears an obvious win. By exploring more hypotheses, massive search is less likely than greedy search to miss the best hypotheses. Surprisingly, the statistics community has long believed that evaluating more hypotheses will result in learning a hypothesis with poor predictive ability. The problem is the more hypotheses analyzed, the greater the chance of finding a *fluke theory*, a hypothesis that accidentally fits the data but does not identify a real pattern. Quinlan and Cameron-Jones [1995] call this phenomenon *oversearching* and have demonstrated that, when evaluating rules using a ranking function called *Laplace accuracy*, the predictive ability of the highest-scoring rule decreases with large amounts of search. Quinlan and Cameron-Jones' results suggest that massive search may not improve inductive performance and present a serious challenge to our central thesis.

The oversearching problem is very similar to the overfitting problem often encountered in machine learning. Overfitting occurs when a complex hypothesis is learned that too closely mimics the training data. While Quinlan and Cameron-Jones argue oversearching is separate from overfitting, we present new results that suggest that the two are closely related. These new results imply that the oversearching effect is a by-product of overfitting and can be reduced by employing standard techniques for

avoiding overfitting. We develop a new variant of Laplace accuracy called *Laplace-depth* that includes a penalty for rule complexity to reduce overfitting. Our results show that Laplace-depth significantly reduces oversearching but does not completely eliminate it.

1.2.3 Applications

The idea of using extensive search for machine learning applies to many learning tasks. As a first step towards showing massive search's wide applicability, we adapt our massive-search algorithm to handle several other machine-learning tasks. Specifically, we show that Brute is easily adapted for classification and data-mining problems.

The extension of Brute to classification is straightforward because there already exists several classification algorithms that make use of a rule-learning component. By replacing the greedy rule-learning component in one of these algorithms with our massive-search algorithm, we can learn better classifiers. Chapter 6 demonstrates that replacing the rule-learning component of CN2 [Clark&Niblett 89] with massive search improves CN2's inductive performance.

The availability of massive search makes it possible to consider alternative control structures for learning classifiers. For instance, while CN2 can make use of a massive-search rule-learning component, it still uses a greedy algorithm for building a classifier from each rule learned. We develop an alternative control strategy that does away with greedy search entirely and has strong theoretical underpinnings.

The extension of Brute to data mining has also proven fruitful. Data mining is the analysis of large databases to extract useful information. While data mining often requires learning descriptive rules, data mining is typically performed using the more readily available classification algorithms. In this thesis, we argue that data mining is best considered a rule-learning problem in which the goal is to find multiple rules rather than the single best. With this improved statement of the data-mining problem, we extend Brute for data mining and achieve a new algorithm that

outperforms existing algorithms on a Boeing manufacturing application.

1.3 Scientific Contributions

The main scientific contributions made by this thesis are summarized below:

- *An Efficient Massive-Search Algorithm for Rule Learning.* The massive-search algorithm we present uses several novel pruning rules and rule-evaluation techniques to achieve a highly-efficient algorithm that can completely search most UCI benchmark databases and several Boeing databases in less than three minutes on a PowerPC-41T workstation.

- *An Alternative Explanation for Oversearching.* This thesis provides empirical evidence that the frequently-encountered problem of evaluating too many hypotheses can be partially attributed to evaluation functions that overfit the data. Using this finding, we have developed improved evaluation functions that substantially reduce the oversearching effect.

- *Demonstration of Improved Inductive Performance using Massive Search.* While previous experiments have failed to benefit from massive search, experiments with our improved evaluation functions demonstrate that massive-search algorithms consistently learn better rules than greedy search. In our experiments, massive search performed better than greedy search in 13 of 18 benchmark databases and performed equally well on the remaining five.

- *Improved Definitions and Algorithms for Data Mining.* By analyzing a Boeing data-mining application, we develop a new definition of the data-mining problem that uncovers some key limitations of existing systems. We use our analysis to develop an improved algorithm based on massive search that outperforms existing algorithms.

- *Improved Algorithms for Learning Classifiers.* As a result of our new evaluation function, we show that replacing the greedy rule-learning component of CN2-like algorithms with massive search improves inductive performance. We also develop a novel classification algorithm that eliminates the need for the greedy covering algorithms used in CN2-like algorithms.

1.4 Organization

Chapter 2 begins by describing the rule-learning problem and demonstrating there is room for improvement over existing greedy algorithms.

Chapter 3 presents Brute, our massive-search algorithm for rule learning. After describing Brute's basic algorithm, the chapter presents Brute's rule-pruning methods and optimizations for rule evaluation. The chapter concludes with an empirical demonstration that massive search is practical on a wide range of benchmark databases as well as on several Boeing manufacturing databases.

Chapter 4 addresses the key question as to whether massive-search algorithms learn better rules. The chapter begins by reproducing Quinlan and Cameron-Jones' experiments showing a degradation in inductive performance with large scale search. It then analyzes when massive search performs poorly and concludes, contrary to Quinlan and Cameron-Jones' analysis, that the oversearching effect is caused by overfitting. The chapter then proposes a new evaluation function to minimize the effects of overfitting and demonstrates that massive search outperforms greedy search.

Having investigated the fundamental questions about massive search, the remainder of the thesis discusses applications of massive search to other learning tasks. Chapter 5 discusses data mining in the context of a Boeing manufacturing application. The chapter begins by formally defining a class of data-mining tasks that previously has been poorly specified in the literature. This definition uncovers several limitations in existing data-mining systems and suggests that data mining is best

considered a rule-learning problem rather than a classification problem. The chapter goes on to describe an extension of Brute's core algorithm designed specifically for data mining. The chapter concludes by showing that this extension of Brute outperforms algorithms currently used for data mining.

Chapter 6 discusses using massive search for classification. The chapter first shows that CN2-like learning algorithms can be improved by replacing their greedy rule-learning component with Brute. However, any such algorithm is limited by CN2's greedy covering algorithm used to form a complete classifier from the rules found by the rule-learning component. The chapter then presents a new classification algorithm that replaces CN2's greedy covering algorithm with a novel, non-greedy covering algorithm that is theoretically motivated.

Chapter 7 concludes by summarizing our results and discussing future work.

Chapter 2

RULE LEARNING

The *rule-learning problem* is the problem of learning a single rule that best describes some portion of the available data. This chapter describes the rule-learning problem and investigates existing solutions. We show that the standard technique for learning rules, greedy search, is limited by its narrow exploration of the search space. This limitation suggests that rule-learning algorithms and the many algorithms that make use of them can be improved by performing more extensive search.

The next section formally defines the rule-learning problem. The following section discusses the instance of rule learning we focus on in this thesis, propositional rule learning. We then discuss existing solutions to the rule-learning problem in Section 2.3 and discuss the relative merits of greedy and massive search.

2.1 Problem Definition

The rule learning problem is the problem of extracting the most informative rule from a database. The inputs and outputs for the rule-learning problem are shown in Figure 2.1. The inputs to the problem are a database language, a database in that language, a set of tests, a goal predicate, and a utility function. The output is a conjunctive rule formed over the test set that maximizes the utility function.

The database $\mathcal{D}$ is a set of positive and negative instances that demonstrate when the goal predicate holds. The goal of rule learning is *not* dependent on the input database $\mathcal{D}$ since the utility function $\mathcal{U}()$ does not take $\mathcal{D}$ as an argument. Successful learning is not defined in terms of the available data but in terms of how well the

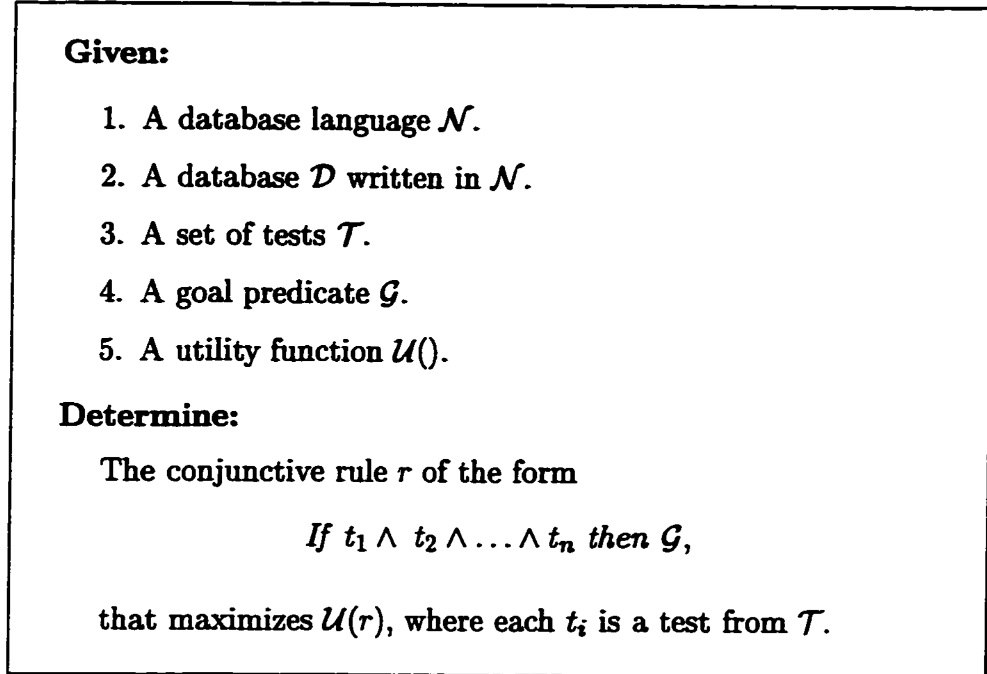


Figure 2.1: High-level description of the rule-learning problem.

rules learned approximate reality. The database provided is used to estimate the true utility of a rule. The quality of this estimate will determine an algorithm's ability to extract good rules.

In order to evaluate a rule-learning algorithm's ability to approximate reality, it is useful to have a model of the physical environment the database represents. Let $\mathcal{I}$ denote the universe of all possible instances. The physical world can be modeled by a probability distribution $\mathcal{P}$ over $\mathcal{I}$ that describes the probability that each instance will occur. We assume $\mathcal{D}$ is representative of the physical environment by assuming the database $\mathcal{D}$ is randomly sampled from $\mathcal{I}$ according to $\mathcal{P}$.

The test set $\mathcal{T}$ and the learning goal $\mathcal{G}$ serve to define the set of rules that the learning algorithm must consider. A test is a condition on the world state. We say that a test t_1 holds for an instance $e \in \mathcal{I}$ if and only if the condition on the world state represented by t_1 is true in e. The goal predicate $\mathcal{G}$ also expresses a condition

on the world state, the condition the user wishes to investigate.

A rule is an expression of the form "*If* $t_1 \wedge t_2 \wedge \ldots \wedge t_n$ *then* $\mathcal{G}$," where each $t_i \in \mathcal{T}$. We define $\mathcal{R}_{\mathcal{T}}$ to be the set of all possible rules defined over $\mathcal{T}$. The goal of learning is to find the rule in $\mathcal{R}_{\mathcal{T}}$ that maximizes the user-provided utility function.

The utility function $\mathcal{U}()$ is used to identify the rules that interest the user. The utility function is a function of a rule's characteristics, including its accuracy, coverage, complexity and evaluation cost. Most of these values are easily evaluated using the syntactic properties of each rule. Accuracy and coverage are unique in that they cannot be evaluated using syntactic properties but require knowledge of the real world.

A rule is a prediction that the rule's consequent will occur when the rule's antecedent holds. The accuracy of this prediction, and thus the accuracy of a rule, is the probability that the consequent holds when the antecedent is true. Let r_{ant} and r_{cons} represent the antecedent and consequent of a rule r. The accuracy $\mathcal{A}(r)$ of the rule r can then be defined in terms of the probability distribution $\mathcal{P}$.

$$\mathcal{A}(r) = \mathcal{P}(r_{cons} | r_{ant}).$$

The coverage or frequency of a rule is the probability the antecedent holds

$$\mathcal{C}(r) = \mathcal{P}(r_{ant}).$$

These measures are both defined using the probability distribution $\mathcal{P}$ which is not available to the learner. In order to evaluate the utility function, the learning algorithm must estimate these values from the database.

The most common utility function and the utility function used throughout this thesis is rule accuracy, $\mathcal{U}(r) = \mathcal{A}(r)$. Rule accuracy is appropriate in situations where rule performance is paramount and other characteristics such as rule coverage and evaluation cost can be safely ignored.

We can now completely describe the task faced by a rule-learning algorithm. A rule-learning algorithm must search the space of rules $\mathcal{R}_\mathcal{T}$ for a rule that is expected to maximize $\mathcal{U}()$. In order to determine whether a rule r is likely to maximize the utility function, the learning algorithm must estimate both the accuracy of a rule $\mathcal{A}(r)$ and the coverage of the rule $\mathcal{C}(r)$. What makes the rule-learning problem difficult is the size of the rule space, $2^{|\mathcal{T}|}$, and computing good estimates for rule accuracy and rule coverage.

2.2 *Propositional Rule Learning*

The description of the rule-learning problem in the last section made no reference to the database language $\mathcal{N}$. The rule-learning problem is a general problem that is independent of the choice of database language. While the main results of this thesis are not dependent on the database language, choosing a specific database language simplifies the discussion. We will use the *attribute-value* representation that is common in the machine-learning literature. In this section we make the attribute-value representation explicit.

An attribute-value language $\mathcal{N}$ is a vector of attributes $[A_0, A_1, \ldots, A_\alpha]$. Each attribute is either one of two types, discrete or numerical. The range of a discrete attribute, $Range(A_i)$, is a finite set of values. The range of a numerical attribute, $Range(A_j)$, is all real numbers between $Min(A_j)$ and $Max(A_j)$ inclusive.

An example in an attribute-value language contains values for each of the attributes. We write an example $e \in \mathcal{I}$ as a vector $[v_0, v_1, \ldots, v_\alpha]$ such that each v_i corresponds to the value of A_i. The value of each attribute is limited by its range; and therefore, each v_i must be a member of $Range(A_i)$.

A sample attribute-value database is shown in Table 2.1. The table shows the database language along with the database itself. The database describes the choice someone might make between playing tennis or racquetball based on weather condi-

tions. This database contains five attributes: *class, outlook, temperature, humidity,* and *windy*. The range for each attribute is given in the table. The database contains several examples of the choice between tennis and racquetball being made. For each example, the value for each of the five attributes is listed.

Attribute-value learning is usually done with a test set determined by the attribute language. For each discrete attribute A_i and for each possible value $v \in Range(A_i)$, the set $\mathcal{T}$ contains the tests $A_i = v$ and $A_i \neq v$. For each numerical attribute A_j and for each possible value $v \in Range(A_j)$, the set $\mathcal{T}$ contains the tests $A_j \leq v$ and $A_j > v$. The goal predicate $\mathcal{G}$ is assumed to be of the form $A_0 = v$. We exclude tests on A_0 from the test set to avoid considering useless rules such as "*If* $A_0 = v$ *then* $A_0 = v$." The test set and goal predicate for our example database is shown in Figure 2.2. The test set shown is unbounded since there are an infinite number of tests for each numerical attribute. We do not list the tests "*windy* $\neq$ *true*" and "*windy* $\neq$ *false*" since these express the same conditions on the world state as "*windy* = *false*" and "*windy* = *true*" respectively. The goal predicate expresses the user's desire to extract information about when tennis is chosen.

2.3 *Existing Solutions*

Since the machine-learning community has not focused on the rule-learning problem, there is no algorithm specifically designed for rule learning. However, many classification algorithms including AQ [Michalski 69], CN2 [Clark&Boswell 91], FOIL [Quinlan 90], Greedy3 [Pagallo&Haussler 90], RIPPER [Cohen 95] and RL [Provost *et al.* 93] make use of a rule-learning algorithm as a component. In this section we investigate the rule-learning components of these algorithms and discuss their limitations.

The rule-learning components of these systems have several similarities. They all operate on attribute-value databases, they all search for the rule with the highest accuracy ($\mathcal{U}(r) = \mathcal{A}(r)$), and they all share a similar structure. Rather than present

Table 2.1: Sample database language and database describing the choice between playing racquetball and tennis.

Database Language

Attribute	Range
class	{*tennis, racquetball*}
outlook	{*sunny, overcast, rain*}
temperature	$\{t \mid -50 \leq t \leq 150\}$
humidity	$\{h \mid 0 \leq h \leq 100\}$
windy	{*true, false*}

Database

Class	*outlook*	*temperature*	*humidity*	*windy*
tennis	*sunny*	75	70	*true*
racquetball	*sunny*	80	90	*true*
racquetball	*sunny*	85	85	*false*
racquetball	*sunny*	72	95	*false*
tennis	*sunny*	69	70	*false*
tennis	*overcast*	72	90	*true*
tennis	*overcast*	83	78	*false*
tennis	*overcast*	64	65	*true*
tennis	*overcast*	81	75	*false*
racquetball	*rain*	71	80	*true*
racquetball	*rain*	65	70	*true*
tennis	*rain*	75	80	*false*
tennis	*rain*	68	80	*false*
tennis	*rain*	70	96	*false*

Test Set

$outlook = sunny$, $outlook = overcast$, $outlook = rain$,
$outlook \neq sunny$, $outlook \neq overcast$, $outlook \neq rain$,

$Temperature \leq -50 \cdots Temperature \leq 150$,
$Temperature > -50 \cdots Temperature > 150$,

$Humidity \leq 0 \cdots Humidity \leq 100$,
$Humidity > 0 \cdots Humidity > 100$,

$windy = true$, $windy = false$

Goal Predicate

$class = tennis$

Figure 2.2: Tests and goal predicate for the sample database in Table 2.1. The test set includes all equality and inequality relations over the discrete attributes and all less-than-or-equal and greater-than relations over the numerical attributes. Since the range of the numerical attributes is a real interval, the number of numerical tests is unbounded. The goal predicate is an equality relation over the first attribute. The goal predicate for this domain expresses the desire to learn when tennis is chosen.

each algorithm individually, we first present an idealized algoritm that emphasizes the algorithms' similarities. We then describe differences between the idealized algorithm and these actual systems.

There are two problems an algorithm must address to solve the rule-learning problem. The first is how to estimate a rule's accuracy and coverage. The second is how to find the best rule according to the chosen estimation function given the exponential size of the hypothesis space. The next two sections presents how our idealized algorithm solves each of these problems. The third section discusses the differences between our idealized algorithm and real systems.

2.3.1 Estimating Accuracy and Coverage

Rule accuracy and coverage must be estimated from the available data. We first discuss the estimation of rule accuracy. Rule accuracy is the probability that a rule's consequent is true given that its antecedent holds.

The simplest estimate for rule accuracy is ***data accuracy*** or how often the rule holds in the database. Let $E(r)$ denote the subset of $\mathcal{D}$ for which the antecedent of rule r holds. Let $E_+(r)$ denote the subset of $E(r)$ for which the consequent of r also holds. Data accuracy $\mathcal{A}_\mathcal{D}$ is defined as follows:

$$\mathcal{A}_\mathcal{D}(r) = \frac{E_+(r)}{E(r)}.$$

Data accuracy assumes $E(r)$ is a random sample and estimates $\mathcal{P}(r_{ant}|r_{cons})$ using sample means. The difficulty with using data accuracy to predict rule accuracy is that the size of each sample, $|E(r)|$, varies with each rule. The sample mean for both a rule that perfectly covers one example and for a rule that perfectly covers 1,000 examples is 100%. However, the sample variance for the rule matching 1,000 examples is much lower and therefore the estimate for this rule is more reliable. It is desirable for an estimation function to take both data accuracy and data coverage into account so that the rules learned are both accurate and reliable.

A better estimate for rule accuracy can be obtained using a Bayesian analysis. Assume the probability that a rule has a particular accuracy forms a probability distribution. Since we know nothing about this distribution before seeing any data, we assume that all rule accuracies are equally probable. Given these two assumptions, we can compute the posterior distribution on rule accuracies after seeing the data. The mean of this posterior distribution serves as a good estimate for rule accuracy.

This estimate is known as *Laplace accuracy* and has been used in several learning algorithms [Niblett 87, Clark&Boswell 91, Smyth&Goodman 91, Webb 93]. Laplace accuracy can be calculated using the following formula:

$$\mathcal{L}(r) = \frac{|E_{+}(r)| + 1}{|E(r)| + 2}.$$

See Niblett [1987] for a derivation.

Laplace accuracy has the desirable property of taking into account both accuracy and coverage when estimating rule accuracy. The exact tradeoff between accuracy and coverage is shown in the contour graph of Figure. 2.3. The axes of the graph are data accuracy and data coverage. The intensity of each point is proportional to the Laplace accuracy of the rule for that data accuracy and data coverage. The lighter the point, the higher the value of the Laplace estimate. Each contour line separates the graph into regions with similar Laplace accuracies.

The graph shows that for rules with greater than 50% data accuracy, the accuracy estimate rises with increased coverage. This can be seen by drawing a horizontal line for a fixed accuracy and noting that Laplace accuracy increases as you move along the line from left to right. Similarly, for a fixed data coverage, the estimate strictly improves with higher accuracy. Both of these properties are what we would expect for any good estimation function. We call any estimation function that obeys these properties *rational.*

The Laplace accuracy function is not rational for rules below 50% data accuracy. This occurs because of the assumptions that underlie Laplace accuracy. We assumed

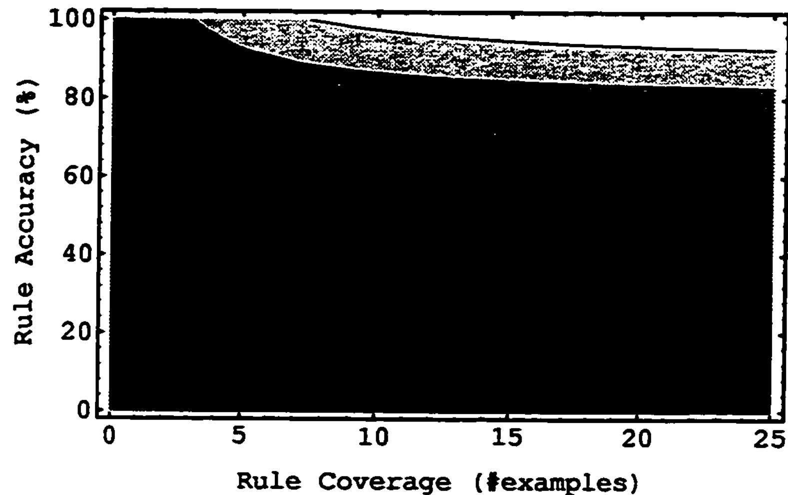


Figure 2.3: Contour graph of Laplace accuracy. The intensity of each point is proportional to its Laplace accuracy. Lighter colors indicate higher Laplace estimates. The contour lines separate the graph into regions with similar Laplace accuracies.

a prior distribution in which all rule accuracies are equally likely. The mean of this prior distribution is 50%. If we find a rule that is less than 50% accurate but with low coverage, we are likely to keep our belief that the rule's actual accuracy is near 50%. We will only lower our accuracy estimate if we are given enough examples to convince ourselves that the rule's accuracy is actually lower than 50%. Hence, Laplace accuracy is symmetric around its midline.

This symmetry is not a problem when learning rules with greater than 50% accuracy. However, it is a problem if we want to learn rules with less than 50% accuracy. This problem can be solved by making an assumption about the mean of the prior distribution. The resulting estimation function will be rational for any rule above the mean we select. A natural choice for the mean of the prior distribution is the frequency the goal predicate is true in the input database. This value is equivalent to the data accuracy of the rule "*If true then* $\mathcal{G}$" that always predicts the goal predicate holds. We refer to this special rule as the *empty rule* and we refer to the accuracy of this rule as the *base accuracy*, $\mathcal{A}_{base}$. The improved formula for Laplace accuracy

implied by this additional assumption is as follows:

$$\mathcal{L}(r, \mu, k) = \frac{|E_+(r)| + k \times \mu}{|E(r)| + k},$$

where k controls the variance of the prior distribution. We will normally set $\mu = \mathcal{A}_{base}$ and $k = 2$, which gives the same variance as the original Laplace function.

While this function has not been used extensively in the machine-learning literature, we believe this modified form of Laplace accuracy is a better function because it correctly handles rules with less than 50% accuracy. This function is commonly used in the statistical literature [Good 65, Smyth&Goodman 91]. Most of the results presented in this thesis are not dependent on the choice of estimation function. Some sections do assume that the estimation function is rational over the range of interest. However, it is useful to have a specific utility function for presenting examples and running experiments. We will use the extended version of Laplace accuracy for this purpose throughout the thesis. It is a good choice because it has a strong theoretical basis, it is very similar to what is used in many rule-learning algorithms, and because with this one fix, it meets the requirements of a good estimation function. In the remainder of the thesis, the term Laplace accuracy will be used to refer to the extended version of Laplace accuracy that includes the correction described above.

The estimation of rule coverage is significantly easier than that of rule accuracy. Rule coverage $\mathcal{C}(r)$ can be reliably estimated using data coverage $\mathcal{C}_\mathcal{D}(r) = |E(r)|$. This estimate is reliable since the sample size for each estimated rule is the same, $|\mathcal{D}|$.

2.3.2 Idealized Algorithm

The second problem any solution to the rule-learning problem must address is how to find the rule with the highest estimated accuracy. What makes this problem difficult is that the hypothesis space grows exponentially with the size of the test set. For a real-world problem with as little as 1,000 tests, the size of the hypothesis space

```
FUNCTION GreedyLearn(Goal, Tests, Database):RULE
    BestRule = EmptyRule(Goal)
    BestScore = ComputeScore(BestRule, Database)
    REPEAT
        Current = BestRule
        NewRules = FALSE
        FOREACH Test IN Tests DO
            NewRule = AddConjunct(Current, Test)
            NewScore = ComputeScore(NewRule, Database)
            IF NewScore > BestScore THEN
                NewRules = TRUE
                BestScore = NewScore
                BestRule = NewRule
            ENDIF
        END
    UNTIL NewRules = FALSE
    RETURN BestRule
END
```

Figure 2.4: Greedy algorithm commonly used to solve the rule-learning problem. The algorithm builds a rule one conjunct at a time. At each iteration through the main loop, the conjunct added is the one that produces the rule with the highest score. The algorithm stops adding conjuncts when adding additional conjuncts does not improve the current rule's score.

is $2^{1000} \approx 10^{301}$. It would take 200 years to evaluate each of these rules on a 500 MHz processor if we could evaluate a rule every cycle. It is clearly not feasible to enumerate all hypotheses and return the rule with the highest accuracy. Instead, existing systems use greedy search to find high-scoring rules in polynomial time.

Figure 2.4 presents the basic greedy algorithm for rule learning. The algorithm builds a rule one conjunct at a time. During each iteration, the algorithm adds the conjunct that produces the rule with the highest estimated accuracy. The algorithm continues adding conjuncts to the rule until there are no tests that provide an improvement.

As described, the greedy-search algorithm will not terminate if the database con-

tains numerical attributes because the test set defined for numerical attributes is unbounded. While there are an infinite number of tests for a numerical attribute A_i, these tests can make only a finite number of distinctions among the examples of $\mathcal{D}$. Let c_1 and c_2 denote any two values of A_i that appear in $\mathcal{D}$ such that no example in $\mathcal{D}$ has a value of A_i between c_1 and c_2. Assume $c_1 < c_2$ and consider the infinite set of tests $A_i \leq v_j$ where $c_1 \leq v_j < c_2$. Each of these tests match the same examples in $\mathcal{D}$ since there are no examples with values of A_i between c_1 and v_j. Any of these tests can be substituted for one another without affecting a rule's estimated accuracy. A rule-learning algorithm can consider only one test from this infinite set and still guarantee finding the highest-scoring rule. For the rest of the thesis, we assume the test set $\mathcal{T}$ only includes tests of the form $A_i \leq c$ and $A_i > c$ for values c that appear in $\mathcal{D}$.

Greedy search is very efficient because it only evaluates a small subset of all rules. The main loop is executed for every test that is added to the final rule. For each execution of the main loop, the inner loop is executed for each of the tests in the test set. If we let l denote the length of the rule learned and we let $|\mathcal{T}|$ denote the size of the test set, then the number of rules evaluated by the greedy algorithm is $l \cdot |\mathcal{T}|$. This is substantially smaller than the $2^{|\mathcal{T}|}$ rules required by simple enumeration and is what makes greedy search efficient.

Greedy search, even though it does not explore the entire search space, produces optimal results for many search problems [Cormen *et al.* 90]. Whether greedy search will produce optimal solutions for rule learning depends on the structure of the optimization problem. Unfortunately, the structure of the rule-learning search space prevents greedy search from being optimal.

Once a greedy algorithm adds a conjunct to its current rule, the conjunct will never be removed. For a greedy algorithm to find the highest-scoring rule, it must not add any tests other than those in the highest-scoring rule. For this to happen, the highest-scoring conjunct at each stage of the search must be one of the tests

from the highest-scoring rule. Unfortunately, this condition is often not met in rule-learning problems. An extreme example of this difficulty occurs when learning from a database describing an exclusive-or relationship. Consider a database with five binary attributes — A, B, C, D, and G — and a goal predicate $G = true$ that is selected from the following probability distribution:

$$\mathcal{P}(G|A,B,C,D) = \begin{cases} 100\% & \text{when } A = true \wedge B = false, \\ 100\% & \text{when } A = false \wedge B = true, \\ 0\% & \text{otherwise.} \end{cases}$$

This database describes an exclusive-or relationship between A and B: G is true when either A holds or B holds but not when both A and B hold. Assume $\mathcal{D}$ is a database randomly chosen according to $\mathcal{P}$ above. The two rules with the highest accuracy are

$$\begin{array}{l} \textit{If } A = true \wedge B = false \textit{ then } G = true \\ \textit{If } A = false \wedge B = true \textit{ then } G = true, \end{array}$$

and we would expect one of them to have the highest estimated accuracy on $\mathcal{D}$. Assume that the first rule has the highest estimated accuracy. This will be the rule we expect our learning algorithm to find. For a greedy algorithm to find this rule, one of the two rules

$$\begin{array}{l} \textit{If } A = true \textit{ then } G = true \\ \textit{If } B = false \textit{ then } G = true \end{array}$$

must have the highest score of all single-conjunct rules. If we look at the probability distribution for all single-conjunct rules,

$$\begin{array}{rclcrcl} P(G = true \mid A = true) & = & 50\% & & P(G = true \mid C = true) & = & 50\% \\ P(G = true \mid A = false) & = & 50\% & & P(G = true \mid C = false) & = & 50\% \\ P(G = true \mid B = true) & = & 50\% & & P(G = true \mid D = true) & = & 50\% \\ P(G = true \mid B = false) & = & 50\% & & P(G = true \mid D = false) & = & 50\%, \end{array}$$

we see that each of these rules are equally likely to be the best rule on $\mathcal{D}$. The rule that actually has the highest score will depend on sampling variations and is unlikely to be either the rule containing $A = \mathit{true}$ or $B = \mathit{false}$.

While it is clear that there are some problems for which greedy search will not find the highest-scoring rules, it is not clear how often these problems occur and how suboptimal are the solutions in practice. We can answer these questions by analyzing the performance of greedy algorithms on benchmark databases and comparing the rules learned to the best possible rules. Table 2.2 presents several benchmark databases from the UCI Machine Learning Data Repository [Murphy 94] that will be used throughout this thesis for analyzing learning algorithms.

Table 2.3 shows the results of running greedy search on the benchmark databases. The table shows the Laplace accuracy of the highest-scoring rule as well as the Laplace accuracy of the rules learned by greedy search. The results for each database were calculated by randomly sampling half the data and running a greedy and exhaustive search on the sample to find the highest-scoring rule. The results show the average score of the highest-scoring rule found by greedy and exhaustive search averaged over ten iterations.

Greedy search did not find the highest-scoring rule for any of the databases, although it was within 0.1 percentage points for two of the eighteen databases. For six databases, the best rule found by greedy search was more than five percentage points below the highest-scoring rule. The largest difference occurred on the Glass database in which the rule learned by greedy search was almost fifteen percentage points below the highest-scoring rule.

The shortcomings of greedy search is emphasized when considering error rates. A rule's error rate is how often it makes an incorrect prediction. A rule's error rate is $1 - \mathcal{A}(r)$ and can be estimated using $1 - \mathcal{L}(r)$. The last column of Table 2.3 shows the ratio of the estimated error for greedy search to the estimated error of exhaustive search. The rules learned by greedy search have a 50% higher estimated

Table 2.2: Description of benchmark databases from the UCI data repository. The values V_d and V_c represent the average number of values per discrete and numerical attribute respectively. Their importance shall be discussed in Chapter 3.

Database	Examples	Goal Classes	Discrete Attributes	V_d	Numerical Attributes	V_c
Autos	205	6	10	6	15	61
Cancer	286	2	9	5	—	—
Chess	3,196	2	36	2	—	—
Credit	690	2	9	4	6	188
Diabetes	768	2	—	—	8	157
Glass	214	7	—	—	9	104
Hepatitis	155	2	13	2	6	54
Iris	150	3	—	—	4	31
Lymphography	148	6	18	3	—	—
Monk1	556	2	6	3	—	—
Monk2	556	2	6	3	—	—
Monk3	556	2	6	3	—	—
Mushroom	8,416	2	22	5	—	—
Promoters	106	2	57	4	—	—
Soybean	683	19	35	3	—	—
Thyroid	3,163	2	18	2	7	165
Tumor	339	22	17	2	—	—
Voting	435	2	16	2	—	—

Table 2.3: Comparison of greedy search and exhaustive search for rule learning. The table shows the Laplace accuracy for the best rule found using each technique. When comparing learning algorithms, the relative error rate is often the most important criteria. The estimated error rate for a rule r is $1 - \mathcal{L}(r)$. The last column shows the ratio of the error rate for greedy search to the error rate for exhaustive search. On average, the error rate of greedy search is 1.8 times higher than the error rate of exhaustive search.

	Exhaustive Search	Greedy Search	
Dataset	Laplace Estimate	Laplace Estimate	Error Estimate Increase
Autos	90.6	77.1	2.4
Cancer	95.3	89.2	2.3
Chess	99.7	97.9	8.0
Credit	98.8	95.7	3.7
Diabetes	98.3	94.8	3.1
Glass	90.1	75.3	2.5
Hepatitis	96.4	94.1	1.6
Iris	94.8	90.3	1.9
Lymphography	93.9	88.8	1.8
Monk1	96.9	95.1	1.6
Monk2	93.6	89.0	1.7
Monk3	96.8	93.8	1.9
Mushroom	99.9	99.8	4.3
Promoters	95.8	92.9	1.7
Soybean	90.0	84.9	1.5
Thyroid	99.8	99.7	1.8
Tumor	70.8	60.4	1.4
Voting	98.9	98.8	1.1
	94.5	89.9	1.8

error on all but two of the databases. On seven of the databases, the rules learned by greedy search have more than twice the estimated error. From this experiment, we conclude that greedy search cannot always find the highest-scoring rules and that the use of more extensive search has the promise of finding rules with higher estimated accuracies.

We cannot conclude from this experiment that the rules learned using massive search are better than those learned using greedy search. For the rules to be better, they must have higher accuracy. The results above only show that exhaustive search finds rules with substantially higher *estimated* accuracies. How well this improvement in estimated accuracy transfers to improvements in true accuracy is the topic of Chapter 4.

2.3.3 Real Systems

Greedy3 [Pagallo&Haussler 90] and RIPPER [Cohen 95] use the greedy-search algorithm presented in the previous section. AQ [Michalski 69], CN2 [Clark&Boswell 91] and RL [Provost *et al.* 93] use an extension of the basic greedy-search algorithm called *beam* search.[1] A beam search maintains a list of the B best rules at each stage of its execution. During each iteration, a beam search evaluates all single-test extensions to its current B rules and selects the best B extensions to be passed as input to the next iteration. A beam search improves on greedy search by giving the learning system B more chances to find the highest-scoring rule. Beam search is practical because the added search complexity grows linearly with B.

FOIL [Quinlan 90] uses a similar extension to greedy search called checkpointing. Checkpointing is designed to improve greedy search when learning exact descriptions. The greedy search algorithm described earlier can return rules with less than 100% data accuracy. This is undesirable when learning exact descriptions since a rule that

[1] Also known as *star* search.

has less than 100% data accuracy cannot be 100% accurate. Checkpointing helps alleviate this problem by performing additional search when a rule with less than 100% data accuracy is learned using standard greedy search. Checkpointing systems remember any point in which the greedy algorithm chose to add one test when another test of similar estimated accuracy could have been added. If the normal greedy search fails to find a rule with 100% data accuracy, then it will back up to the points it has remembered and try the next best tests. The amount of additional search done by checkpointing is limited by restricting the system to a fixed number of checkpoints. Like beam search, checkpointing provides additional opportunities to find better rules.

Each of these extensions to greedy search adds a bounded amount of additional search in the hope of finding rules with higher estimated accuracy. These systems only add a bounded amount of search because it is assumed that a more extensive search will be too costly. The results of Table 2.3 suggest that performing large scale search, if practical, will result in better performance. In the next chapter we present Brute, a rule-learning system that aggressively searches for the rule with the highest estimated accuracy. Brute's advanced search techniques and efficient implementation allow it to search a substantially larger portion of the search space than any existing system. Brute can perform a complete search on many databases. Brute's extensive search results in finding rules with substantially-higher estimated accuracy.

Chapter 3

BRUTE

Brute conducts a massive search of rule space to avoid the pitfalls of greedy search. Brute efficiently searches this exponential space using advanced search techniques and rule-pruning strategies that minimize the number of rules it needs to evaluate. This chapter presents Brute's search algorithm and analyzes its efficiency. The chapter begins with a high-level discussion of Brute's search algorithm and is followed by sections detailing Brute's rule-pruning and rule-evaluation strategies. The chapter then discusses Brute's complexity and analyzes its performance on our benchmark databases. The chapter concludes with a discussion of related work.

3.1 Basic Algorithm

Brute performs a depth-first search of the space of conjunctive rules to find the rule that maximizes the user-supplied evaluation function. Depth-first search requires organizing the search space using a tree structure. Brute organizes its search tree using the specialization hierarchy shown in Figure 3.1. To save space, the figure shows only the antecedent of each rule. The hierarchy shown is for a domain containing four tests t_1 through t_4. The children of a rule are all specializations formed by adding a single test to the rule's antecedent.

A specialization hierarchy has two advantages. First, the tree is easily generated recursively since each node differs from its parent by a single test. Second, as we will discuss in Section 3.3, specialization hierarchies make it easier to evaluate rules. Greedy search uses the same hierarchy; however, greedy search considers rules from

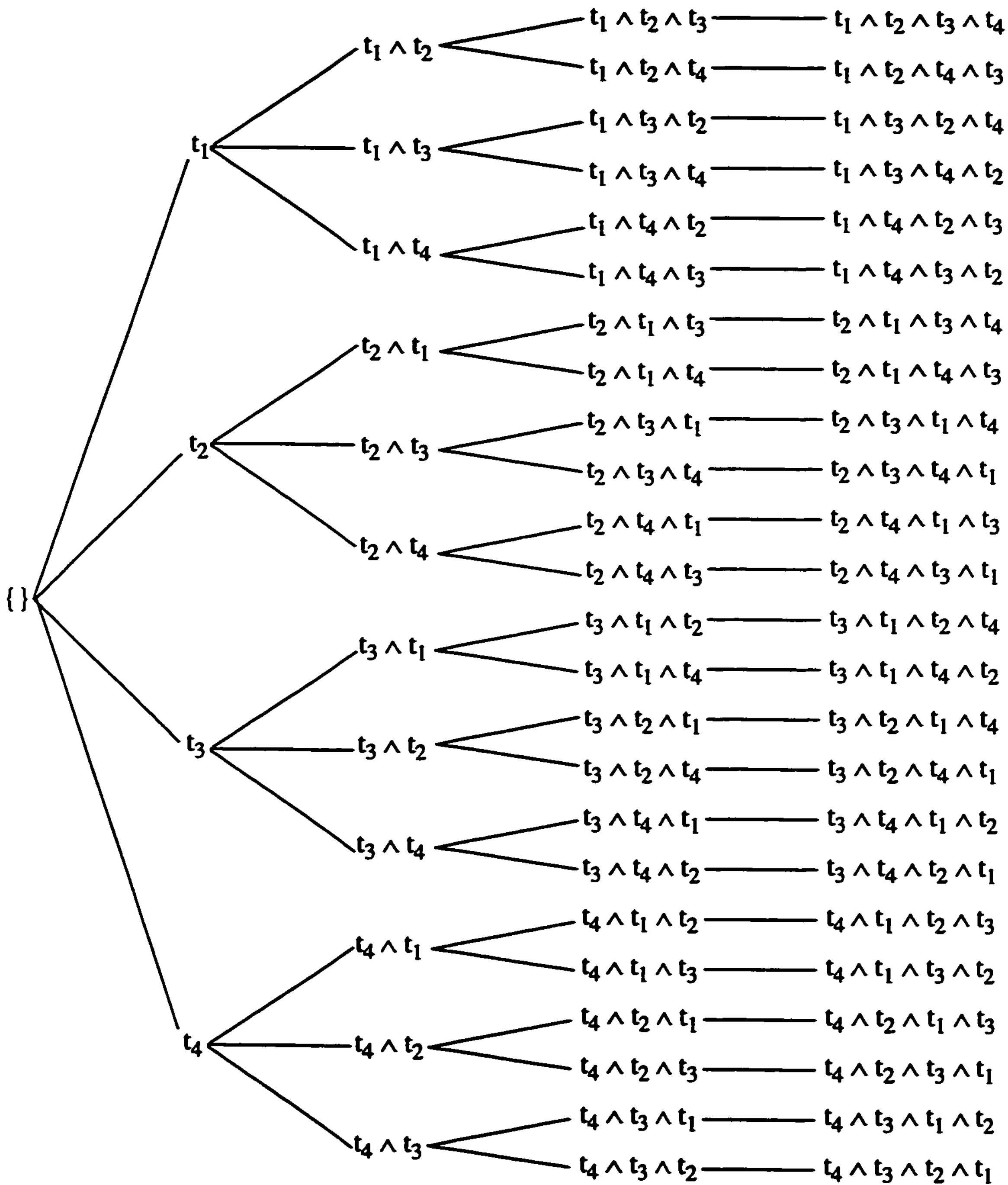


Figure 3.1: Specialization hierarchy used to organize the hypothesis space for depth-first search. Each node represents a single rule whose antecedent contains the labeled conjunction. While this hierarchy is easy to generate and allows for efficient rule evaluation, it contains many duplicate rules.

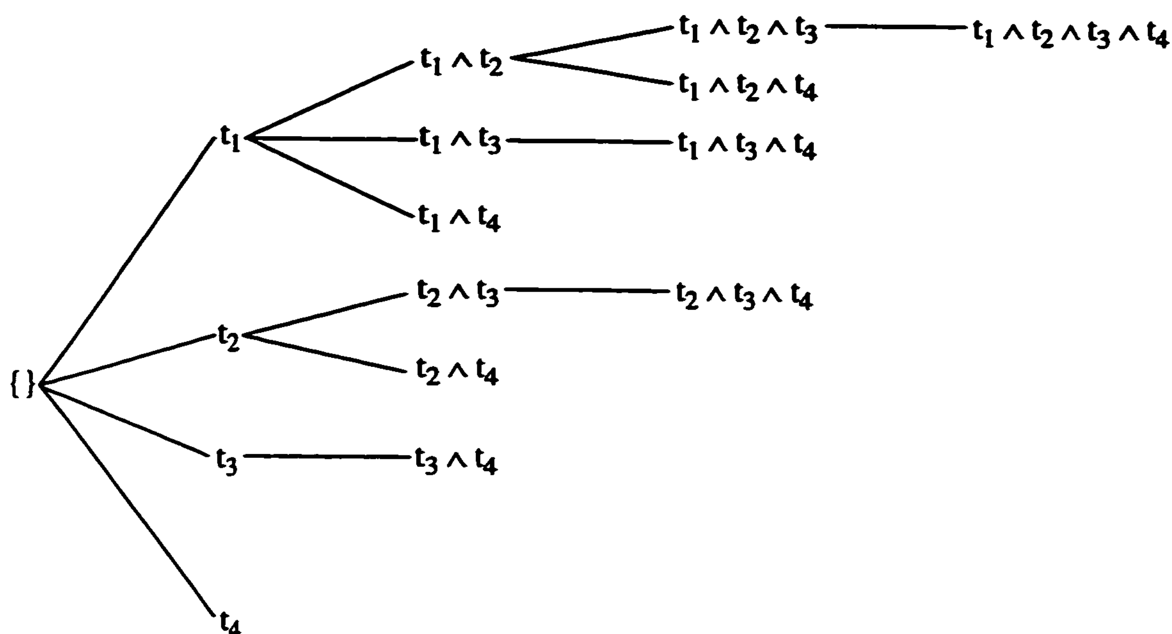


Figure 3.2: Improved specialization hierarchy with no duplicates. Duplication is avoided by only including rules whose tests appear in ascending order.

only one branch of the tree.

The difficulty with the hierarchy of Figure 3.1 is that several nodes contain logically-equivalent rules. Both the conjunctions $t_1 \wedge t_2$ and $t_2 \wedge t_1$ appear in the search tree even though they are logically equivalent. For each rule of length l, there is a node for each of the $l!$ different orderings of the rule's tests. Using this ordering for learning results in much wasted effort.

A learning algorithm is *systematic* if it considers each logically-distinct rule exactly once. Systematicity is desirable since it prevents an exponential amount of repetition. Systematicity can be achieved by imposing an ordering on the tests and never adding a rule whose tests appear in a different order. Figure 3.2 shows how requiring tests to appear in numerical order generates a specialization tree with no repetitions.

Brute explores the search space of Figure 3.2 using depth-first search. A simplified version of Brute's search algorithm is shown in Figure 3.3. The details missing from

this description will be described in the remainder of the chapter. The procedure MassiveSearch() is recursive; every node in the search tree is processed in an identical manner. MassiveSearch() steps through each test to be added to the current rule. For each test, a new rule is formed with this test added. The score of each new rule is computed, and the rule with the best accuracy is remembered. The algorithm then calls PruneRules() to determine which of the new rules can be pruned. The conditions for pruning subtrees are described in Section 3.2. MassiveSearch() is called recursively to process each unpruned rule's specializations.

MassiveSearch() uses a unique method to ensure systematicity. Rather than using an explicit ordering function, it uses the order in which rules are processed by the second FOREACH statement. The tests passed to each recursive call of MassiveSearch() are those tests previously processed by the second FOREACH loop. This ordering guarantees that no two descendants of the current node will be logically equivalent. This is interesting because it does not require the orderings for each recursive call to be the same. If the second FOREACH statement processed the tests randomly, systematicity will still be assured. We will use this later to improve search efficiency.

Brute's efficiency is important because it determines how much of its hypothesis space it can explore in the time it has allotted. The computational cost of Brute is the number of times MassiveSearch() is called multiplied by the cost of each call:

$$O(\mathit{RulesVisited} \times \mathit{CostOfVisit}).$$

We can improve the efficiency of Brute by either reducing the number of rules visited or by reducing the cost of each call to MassiveSearch. These are the topics of the next two sections.

3.2 Rule Pruning

Rule pruning removes portions of the search tree which can be proven not to maximize the evaluation function. Rule pruning speeds the search by reducing the number of

```
PROCEDURE MassiveSearch(CurrentRule, Tests, Database)
    NewRules = ∅
    FOREACH Test IN Tests DO
        NewRule = AddConjunct(CurrentRule, Test)
        ComputeScore(NewRule, Database)
        IF NewRule.Score > BestRule.Score THEN
            BestRule = NewRule
        ENDIF
        NewRules = NewRules ∪ NewRule
    END
    PruneRules(NewRules)
    NewTests = ∅
    FOREACH Rule IN NewRules DO
        MassiveSearch(NewRule, NewTests, Database)
        NewTests = NewTests ∪ LastTestAdded(NewRule)
    END
END

FUNCTION Brute(Goal, Tests, Database):RULE
    BestRule = EmptyRule(Goal)
    MassiveSearch(BestRule, Tests, Database)
    RETURN BestRule
END
```

Figure 3.3: Simplified version of Brute's massive-search algorithm. Brute conducts a depth-first search of the search tree shown in Figure 3.2. Brute's search is systematic and visits each rule exactly once. Brute avoids searching all rules by pruning portions of the search space that it can prove do not contain the best rule.

rules that must be evaluated. Since the rules removed by rule pruning are proven not to be maximal, rule pruning improves performance without affecting the quality of the learned rules.

Rule pruning makes massive search practical. For example, Brute only needs to evaluate 10^4 of the 10^{74} possible rules to find the maximal rule for the Promoters database. Rule pruning improves performance on this database by a factor of 10^{70}. Without rule pruning, complete searches would not be practical.

Pruning is effective because each pruning operation can remove a substantial fraction of the entire search space. Brute's pruning axioms can prove that all of a rule's descendents are non-maximal. When Brute prunes a single-test rule, it does not have to evaluate any other rules containing that test. Each test appears in at least one out of every $V+1$ rules, where V is the number of values in the range of the test's attribute. If we prune p first-level tests with V values each, we reduce the size of the search space by a factor of $(\frac{V+1}{V})^p$. For $V=2$, pruning fifteen first-level tests will reduce the size of the search space by a factor of 438. While this effect is reduced at lower levels of the search space, the overall impact is dramatic. Rule pruning makes it possible for Brute to find a maximal rule by only evaluating a small fraction of the search space.

Brute employs four different pruning strategies: branch-and-bound pruning, subsumption pruning, dynamic reorganization, and depth-bound pruning. The following sections present each of these algorithms in turn. Two of these strategies, subsumption pruning and dynamic reorganization, were adapted from OPUS [Webb 95]. Subsumption pruning and dynamic reorganization are discussed below for completeness and because they are necessary for understanding Brute. The following sections also present Brute's search algorithm since the choice of search algorithm has a large impact on pruning performance.

3.2.1 *Branch-and-Bound Pruning*

Branch-and-bound pruning is a standard technique for increasing the effectiveness of depth-first search. Branch-and-bound pruning computes an upper bound for the rule scores within each subtree. Each upper bound is compared against the score of the current best rule. Any subtree whose computed upper bound is less than the score of the current best rule can be safely removed from the search space.

The upper-bound computation must guarantee that the score of the subtree's rules does not exceed the bound computed. Applying the branch-and-bound framework to rule learning requires identifying a suitable bounding function. We present a general framework for determining upper bounds for *rational evaluation functions*.

Rational evaluation functions were defined in Chapter 2 to obey certain properties we expect of all good evaluation functions. Everything else being equal, an evaluation function should prefer a rule with both higher data accuracy and higher data coverage. This is true regardless of how we wish to tradeoff accuracy and coverage. We formalize this criteria by requiring that rational evaluation functions monotonically increase with both data accuracy and data coverage. However, this requirement says nothing about comparing two rules such that one has higher data accuracy and the other has higher data coverage. While this comparison generally depends on the tradeoff between accuracy and coverage for a particular application, there are some instances we expect all good evaluation functions to handle identically. Consider two rules such that rule r_1 covers 750 examples and is 100% accurate and rule r_2 covers 1,000 examples and is 75% accurate. While both rules correctly predict 750 examples, rule r_1 also incorrectly predicts 250 examples. Clearly, r_1 is preferable since it correctly predicts the same number of examples while not making the mistakes of r_2. However, since one rule is more accurate and the other has higher coverage, the requirements of monotonically increasing in both accuracy and coverage is not sufficient to ensure that rational evaluation functions will choose r_1. We therefore add the requirement

that rational evaluation functions monotonically decrease in the number of covered negative examples. The complete definition of rational evaluation functions is as follows:

Definition 1 *A rational evaluation function is any function*

$$S(|E_+(r)|, |E_-(r)|, X_1(r), X_2(r), \ldots)$$

such that

$$\frac{\partial S(r)}{\partial \mathcal{A}_{\mathcal{D}}(r)} \geq 0, \ \frac{\partial S(r)}{\partial \mathcal{C}_{\mathcal{D}}(r)} \geq 0, \ and \ \frac{\partial S(r)}{\partial |E_-(r)|} \leq 0.$$

This definition captures our intuitions about evaluation functions without making any assumptions about the relative importance of accuracy and coverage. Many of the common evaluation functions for rule learning, including Laplace accuracy [Clark& Boswell 91], mutual information [Clark&Niblett 89] and minimum description length [Quinlan&Rivest 89], are all rational for accuracies above a threshold. These functions should not be used for learning rules below their rationality threshold.[1]

Rationality, while capturing the notion of an appropriate evaluation function, is also useful in computing upper bounds. Rationality guarantees two properties that make computing upper bounds easier: a rule's score monotonically increases with the number of positive examples matched, and a rule's score monotonically decreases with the number of negative examples matched. While the first condition is not explicitly stated in the definition of rationality, it is a direct result of the evaluation function monotonically increasing in both data accuracy and data coverage. These two conditions are useful since all descendents of a rule match a monotonically decreasing number of positive and negative examples. They imply that the score of any rule in a subtree rooted at r cannot have a higher score than that obtained by matching all the positive examples matched by r and none of the negative examples. When the

[1] The algorithms analyzed in this thesis adhere to this constraint.

Table 3.1: Characteristics of common evaluation functions. Theorem 1 provides an upper bound for any rational depth-monotone evaluation function.

Function	Rational?	Depth Monotone?
Laplace accuracy	when $\mathcal{A}_\mathcal{D} \geq \mathcal{A}_{base}$	when $\mathcal{A}_\mathcal{D} \geq \mathcal{A}_{base}$
Mutual information	when $\mathcal{A}_\mathcal{D} \geq 0.5$	when $\mathcal{A}_\mathcal{D} \geq 0.5$
Minimum description length	when $\mathcal{A}_\mathcal{D} \geq 0.5$	when $\mathcal{A}_\mathcal{D} \geq 0.5$
Cost matrix	yes	yes

scoring function depends only on positive and negative coverage, a good upper bound for the subtree r is therefore $S_{ub}(r) = S(|E_+(r)|, 0)$.

The situation is more complicated when the evaluation function takes additional parameters. The above analysis works because each parameter has monotonic derivatives (be it increasing or decreasing) and because each parameter's value changes monotonically with increasing rule length. These properties occur naturally with rational evaluation functions. While these monotonicity properties generally do not hold for other parameters, they do hold for several common parameters such as rule length, rule complexity and rule cost. A depth-monotone evaluation function is any evaluation function for which these monotonicity properties hold for all parameters.

Definition 2 *Let $L(r)$ denote the length of a rule. A depth-monotone evaluation function is any function*

$$S(r) = S(X_0(r), X_1(r), \ldots)$$

that is monotonic in each of its input variables $X_i(r)$, and each input variable $X_i(r)$ is monotonic in $L(r)$.

Table 3.1 lists common evaluation functions and displays their rationality and depth monotonicity. The table includes Laplace accuracy [Clark&Boswell 91], mutual

information [Clark&Niblett 89], minimum description length [Quinlan&Rivest 89], and cost matrices [Pazzani *et al.* 94]. All of these functions are rational and depth monotone above some threshold. Rather than try to compute an upper bound for each function separately, we compute an upper bound that applies to all evaluation functions that are both rational and depth monotone.

Theorem 1 *Let $S(r) = S(E_+(r), E_-(r), X_1(r), X_2(r), \ldots)$ denote a rational depth-monotone evaluation function, Let Descendants(r) denote all the descendants of rule r in the search tree, and let $L(r)$ denote the length of rule r. The value of $S(r')$ where $r' \in$ Descendants(r) is less than or equal to*

$$S(|E_+(r)|, 0, B_1(r), B_2(r), \ldots)$$

where

$$B_i(r) = \begin{cases} \infty & \text{when } \frac{\partial S(r)}{\partial X_i(r)} \geq 0 \text{ and } \frac{\partial X_i(r)}{\partial L(r)} \geq 0 \\ X_i(r) & \text{when } \frac{\partial S(r)}{\partial X_i(r)} \leq 0 \text{ and } \frac{\partial X_i(r)}{\partial L(r)} \geq 0 \\ X_i(r) & \text{when } \frac{\partial S(r)}{\partial X_i(r)} \geq 0 \text{ and } \frac{\partial X_i(r)}{\partial L(r)} \leq 0 \\ 0 & \text{when } \frac{\partial S(r)}{\partial X_i(r)} \leq 0 \text{ and } \frac{\partial X_i(r)}{\partial L(r)} \leq 0. \end{cases}$$

Theorem 1 prescribes how to compute a good upper bound for most evaluation functions. For Laplace accuracy, the theorem provides the following formula:

$$\mathcal{L}_{ub}(r) = \mathcal{L}(|E_+(r), 0) = \frac{|E_+(r)| + k \times \mathcal{A}_{base}}{|E_+(r)| + k}.$$

Brute prunes any subtree rooted at r such that the value of $\mathcal{L}_{ub}(r)$ is less than the score of the current best rule.

3.2.2 Subsumption Pruning

Subsumption pruning removes any subtree for which another subtree exists containing strictly better rules. While branch-and-bound pruning removes subtrees based on

their performance relative to the current best rule, subsumption pruning removes subtrees based on their performance relative to other subtrees.

Subsumption pruning removes the subtree below any rule for which there exists another rule that covers a superset of its positive examples and a subset of its negative examples. The intuition behind subsumption pruning is best illustrated by considering the special case of two rules, r_1 and r_2, matching exactly the same set of training examples:

$$\begin{array}{ll} r_1\text{:} & \textit{If } A = a \textit{ then class} = pos \\ r_2\text{:} & \textit{If } B = b \textit{ then class} = pos, \end{array}$$

Since the tests $A = a$ and $B = b$ match the same set of examples, they are functionally equivalent and can be used interchangeably in rules without affecting rule score. This implies that each specialization of r_2 will have the same accuracy as the corresponding specialization of r_1. As a result, it is not necessary to search both subtrees since the corresponding rules in each subtree have the same score.

The general case of subsumption pruning can be understood by extending our example. If the rule r_1 covered a superset of the positive examples covered by r_2 but the same negative examples, then interchanging the test $A = a$ for $B = b$ can only increase the number of positive examples covered by the rule. Furthermore, if r_1 covered a superset of the positive examples covered by r_2 and a subset of its negatives, then interchanging the test $A = a$ for $B = b$ can only increase positive coverage and decrease negative coverage. In either case, every specialization of r_1 will have a higher score than the corresponding specialization of r_2. As a result, it is not necessary to search the subtree below r_2. The following theorem provides the theoretical justification for subsumption pruning:

Theorem 2 *Let $S(E_+(r), E_-(r))$ denote any depth-monotone evaluation function. If r_1 and r_2 are two rules such that*

1. *$E_+(r_1) \supseteq E_+(r_2)$ and*
2. *$E_-(r_1) \subseteq E_-(r_2)$,*

then for all $s_2 \in Specializations(r_2)$ *there exists an* $s_1 \in Specializations(r_1)$ *such that* $S(s_1) \geq S(s_2)$.

Proof:
Let $P = E_+(r_1) - E_+(r_2)$ denote the positive examples matched by r_1 and not r_2. Let $N = E_-(r_2) - E_-(r_1)$ denote the negative examples matched by r_2 and not matched by r_1. Using this notation, we can write $E_+(r_1) = E_+(r_2) \cup P$ and $E_-(r_1) = E_-(r_2) \cap \neg N$. Consider the rule s_2 which is a specialization of r_2. Let $T = Antecedent(s_2) - Antecedent(r_2)$ denote the tests added to r_2 to form s_2. If we let T_+ and T_- denote the positive and negative examples matched by the tests T, then the examples matched by s_2 are $E_+(s_2) = E_+(r_2) \cap T_+$ and $E_-(s_2) = E_-(r_2) \cap T_-$. Let s_1 denote the rule formed by conjoining the tests of T with r_1. If we compare the examples matched by s_1 and s_2, we get the following:

$$\begin{aligned} E_+(s_1) &= E_+(r_1) \cap T_+ \\ &= (E_+(r_2) \cup P) \cap T_+ \\ &= (E_+(r_2) \cap T_+) \cup (P \cap T_+) \\ &= E_+(s_2) \cup (P \cap T_+) \\ &\supseteq E_+(s_2) \end{aligned}$$

$$\begin{aligned} E_-(s_1) &= E_-(r_1) \cap T_- \\ &= (E_-(r_2) \cap \neg N) \cap T_- \\ &= (E_-(r_2) \cap T_-) \cap \neg N \\ &= E_-(s_2) \cap \neg N \\ &\subseteq E_-(s_2). \end{aligned}$$

Since $S()$ is depth monotone, this result implies that $S(s_1) \geq S(s_2)$ and establishes the theorem.□

Extending Theorem 2 to include additional parameters is straightforward.

While Theorem 2 allows us to prune any subsumed rules, pruning all such rules is too expensive. Determining all subsumed rules requires checking all pairs of rules in Brute's search space. It is impractical to visit every rule in Brute's search space, let alone all pairs. Instead, Brute checks for subsumption on rule sets for which the comparison is convenient.

```
PROCEDURE SubsumptionPrune(ParentRule, NewRules)
    FOREACH Rule1 IN NewRules DO
        IF Length(ParentRule.Negatives) = Length(Rule1.Negatives) THEN
            NewRules = NewRules - Rule1
        ELSE
            FOREACH Rule2 IN NewRules DO
                IF Rule1 ≠ Rule2 AND
                    Rule1.Positives ⊇ Rule2.Positives AND
                    Rule1.Negatives ⊆ Rule2.Negatives THEN
                    NewRules = NewRules - Rule2
                ENDIF
            END
        ENDIF
    END
END
```

Figure 3.4: Brute's algorithm for subsumption pruning. The algorithm is a straightforward implementation of Theorem 2.

Brute tests for subsumption among the rules evaluated at each node in the tree. The examples matched by each child are checked against those matched by the other children to determine if any of the children subsume each other. Each child is also checked against its parent. The advantage of this limited subsumption check is that all the information needed for the comparison is readily available.

Figure 3.4 shows Brute's algorithm for subsumption pruning. The algorithm checks subsumption between each child and its parent first, then checks subsumption between each pair of children. The complexity of Brute's subsumption check is $O(|T|^2N)$ because, for each pair of children, all examples must be traversed to check inclusion.

As discussed in Section 3.3, the complexity of processing each node is $O(|T|N)$ without subsumption pruning. Technically, subsumption pruning increases the complexity of node processing to $O(|T|^2N)$. Subsumption pruning does not substantially increase rule-processing costs in practice because detecting non-subsuming rules only

requires finding a single counter example. On our sample databases, subsumption pruning increased rule-evaluation time by a maximum of 16.8% with an average increase of just 6.3%.[2] The reduction in the size of the search space afforded by subsumption far outweighs this additional cost.

The cost of subsumption pruning can also be minimized using specialized algorithms in some situations. Checking subsumption between a rule and its parent can be done by just checking if the two rules match the same number of negative examples. The examples matched by a child rule are a subset of the examples matched by its parent rule. As a result, the child rule is guaranteed to match a subset of the parent's positive examples. The child rule only matches a superset of the parent's negative examples in the case of equality.

The above optimization also applies to checking subsumption between rules involving numerical attributes. Subsumption among all sibling rules for which a test of the form $A_i < t$ has been added can be checked by comparing the number of negative and positive examples matched by each adjacent rule. If either the number of positive examples or the number of negative examples for an adjacent rule is equal, then the rule with the lower score can be pruned by subsumption. This entire operation has complexity $O(V_i)$, where V_i is the average number of values per numerical attribute. This optimization can be quite effective because the number of tests per numerical attribute is often large.

3.2.3 Dynamic Reorganization

The number of rules removed by a pruning operation depends on the number of rules in the pruned subtree. The size of each subtree in turn depends on the organization of the search space. The organization Brute uses to achieve systematicity produces an

---

[2] The additional overhead of subsumption pruning was measured by running Brute on ten random samples from each database and recording the percentage of the overall running time used for subsumption pruning.

unevenly distributed search space with some of a node's subtrees having substantially more rules than others. In the tree previously shown in Figure 3.2, the tree below t_4 has only one example while the tree below t_1 has seven. Pruning the tree below t_1 is much more effective than pruning the tree below t_4.

The ordering used to establish systematicity can also be used to improve pruning performance. Brute's method for ensuring systematicity does not require the same ordering to be used at each node. Pruning performance can be improved by ordering each node's children by decreasing upper bound. This organization guarantees the largest subtree is the one with the smallest upper bound. Since this subtree will be pruned by branch-and-bound pruning before any of its siblings, this organization maximizes the effect of pruning. This technique extends to other pruning rules by assigning a zero upper bound to any pruned rules.

Figures 3.5 and 3.6 show the effect of branch-and-bound pruning when the current best rule has a score of 0.75. The number next to each rule r is the subtree's upper bound, $S_{ub}(r)$. Figure 3.5 shows that the standard lexicographic ordering cannot prune any subtrees because all the rules below 0.75 are leaves. In contrast, Figure 3.6 shows that dynamic reorganization rearranges the search space such that all rules below 0.75 are in the same subtree and therefore will be removed by branch-and-bound pruning. The number of rules that must be evaluated in the reorganized search space is approximately half of what needs to be evaluated without reorganization.

3.2.4 Search Algorithm

The effectiveness of branch-and-bound pruning depends on the order in which Brute explores its search space. Since branch-and-bound pruning only removes subtrees whose computed upper bound is lower than the score of the current best rule, whether a subtree is pruned depends on the quality of the current best rule at the time the decision is made to traverse the subtree. This in turn depends on the order in which the search space is explored.

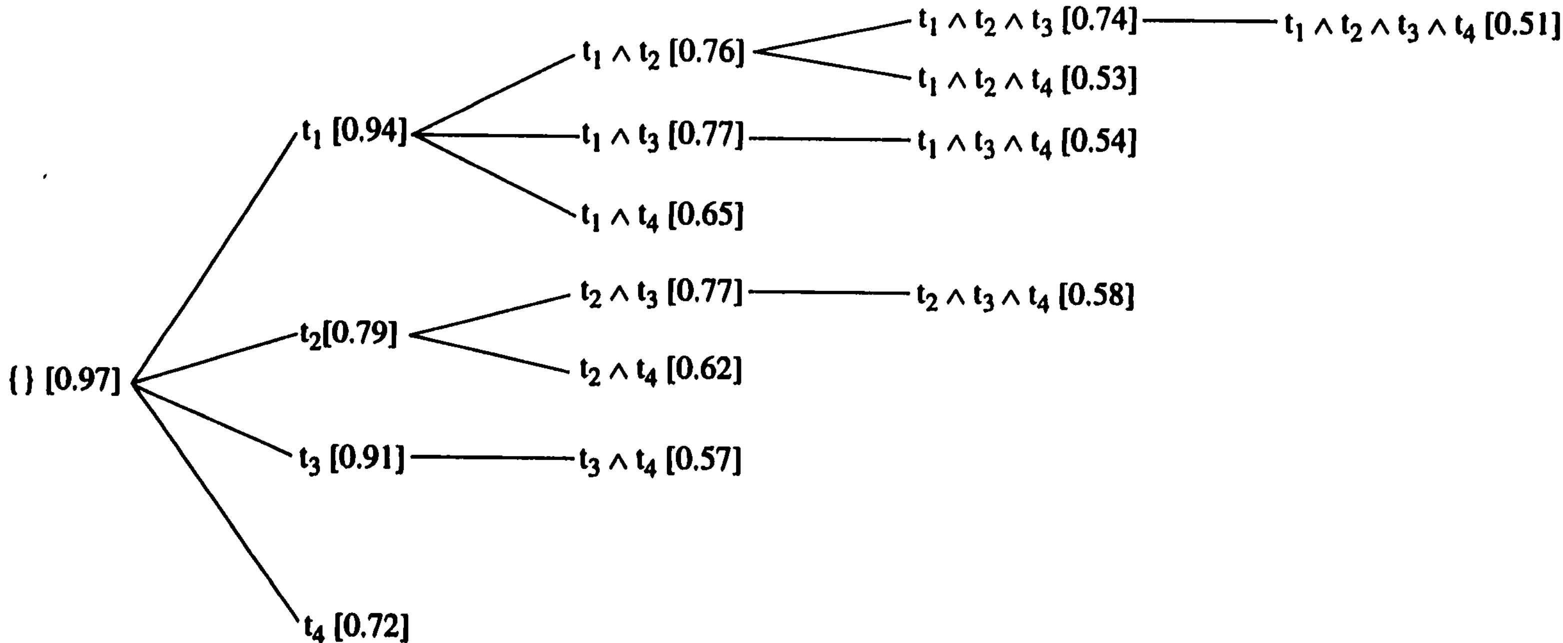


Figure 3.5: Branch-and-bound pruning on a lexicographically-ordered search tree. The upper bounds for each subtree appear next to each rule. When the current best rule has a score of 0.75, none of the subtrees can be pruned even though the tree contains eight rules with lower scores.

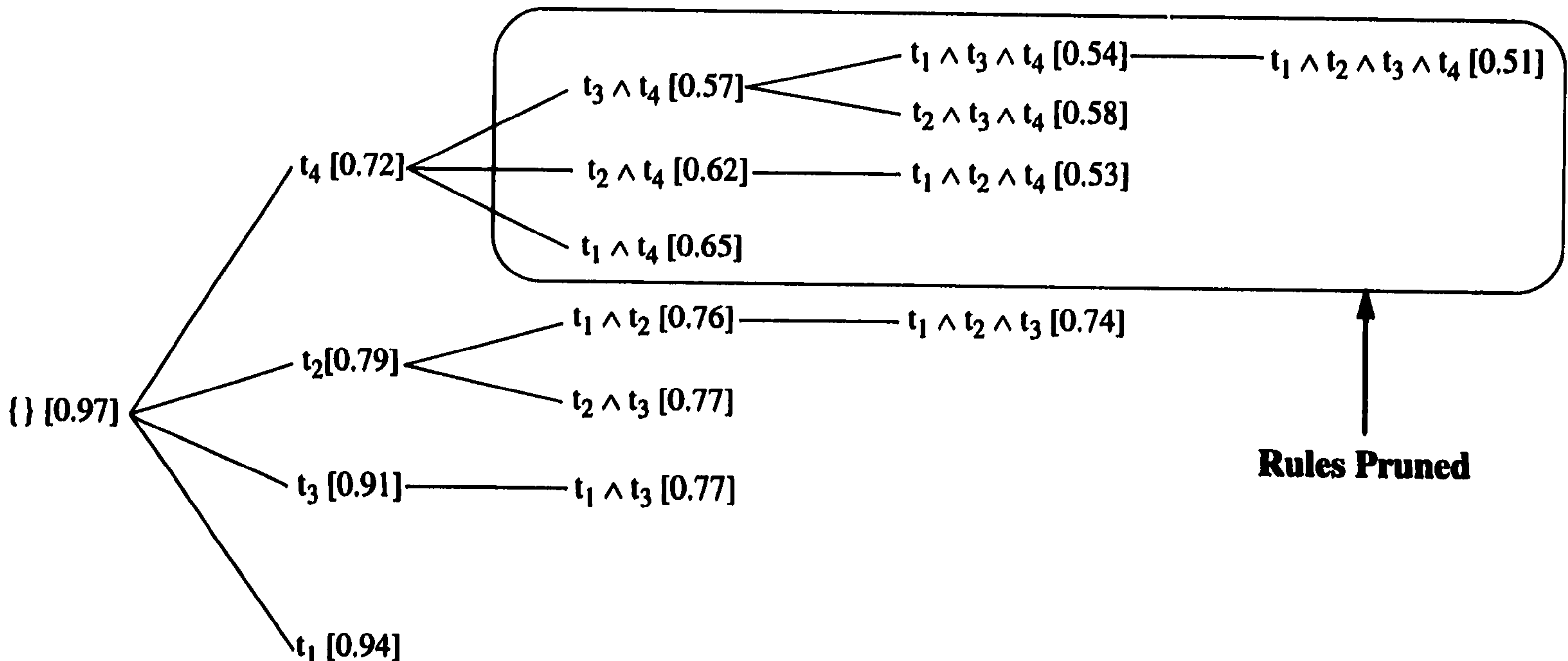


Figure 3.6: Dynamic reorganization sorts each subtree by increasing upper bound to maximize pruning. The enclosed region shows the seven rules pruned by branch-and-bound pruning when the score of the current best rule is 0.75.

Best-first search is an optimal algorithm for maximizing the effect of branch-and-bound pruning. Best-first search avoids exploring any subtree until it can prove that no rule better than its upper bound exists. Best-first search accomplishes optimality by expanding rules in the order of decreasing upper bound. When best-first search starts exploring a subtree, it has already searched all subtrees with an upper bound greater than the upper bound of the current subtree. If a rule existed with a higher score than the current subtree's upper bound, than it would have had to have been in one of the subtrees that had a higher upper bound. Since all these subtrees have already been searched, no such rule can exist

The problem with best-first search is that its memory requirements grow linearly with the size of the search space. Best-first search makes use of a priority-queue to keep track of the unsearched subtrees. As each subtree is expanded, all of its children are added to the priority queue. Brute can process 50,000 rules per second when applied to a database of 1,000 examples. As a result, Brute can add as many as 50,000 rules per second to the priority queue. If each rule requires only 12 bytes of storage, best-first search would require an additional 36 MB's of memory every minute. Best-first search is too memory intensive to be practical.

Depth-first search, while having minimal memory requirements, is not nearly as effective at maximizing the impact of branch-and-bound pruning. Depth-first search explores each subtree recursively without regard to which subtrees contain good rules. As a result, depth-first search often traverses subtrees with low upper bounds early in the search that could have been pruned if their exploration were delayed.

Brute uses an alternative to best-first search called heuristically-guided depth-first search. Heuristically-guided search differs from standard depth-first search by intelligently deciding which subtrees to explore. Brute's heuristically-guided search always explores the subtree with the greatest upper bound. This preference increases the likelihood that Brute will find a high-scoring rule before exploring branches with low upper bounds.

Webb has compared heuristically-guided search and best-first search on several UCI databases [Webb 95]. His results show that, when combined with dynamic reorganization, heuristically-guided search and best-first search have similar pruning performance. In 13 of 14 databases, heuristically-guided search expanded no more than twice the nodes of best-first search, expanding only four times the nodes in the worst case. The performance of Brute is not significantly degraded by using heuristically-guided search while significantly improving Brute's memory requirements.

Another possibility for Brute's search algorithm is a linear-space best-first search algorithm. One such algorithm, iterative deepening [Korf 85], implements best-first search by making multiple calls to a depth-first search engine. The number of calls grows linearly with the number of possible rule scores. Since the number of possible rule scores grows quadratically with the number of examples, iterative deepening is prohibitively expensive.

RBFS [Korf 92] is an improved linear-space best-first search algorithm that requires fewer iterations over the search space. A unique aspect of RBFS is that it only backtracks and evaluates a rule multiple times if the depth-first path it has chosen is non-optimal. Since the performance of heuristically-guided search is close to optimal, RBFS search may not have to revisit many rules. This is an empirical question we leave for future work.

3.2.5 Depth-Bound Pruning

While Brute is very efficient and can completely search a variety of databases, there are many databases for which a complete search is not practical. Massive search is still useful on these databases because the additional search it performs can find better rules than can be found otherwise.

When Brute cannot search all rules in the time available, it must search a subset of the entire space. The subset searched by Brute would ideally be the subset likely to contain the best rules. This can be accomplished using best-first search and stopping

when no more time is available. While the computational cost of best-first search makes this less attractive, there is a more fundamental limitation: it is very difficult to characterize the rules searched.

When Brute conducts a large but incomplete search, the rule found is not guaranteed to maximize the evaluation function. The rule found only maximizes the rules searched. A characterization of Brute's search space is needed to identify the set maximized by the rule returned. Without this information, it is difficult to interpret the search results. An extreme case occurs when the rule returned is judged insignificant by an expert analyst. Without a good characterization of the search space, the search results do not suggest where a significant rule might be found.

Brute uses a depth bound to limit search. A depth bound specifies the maximum length rule to consider. A depth bound has the advantage of searching an easily-characterizable portion of the search space. If a search to a given depth fails to find a significant rule, then any significant rule for this database must be longer than the bound specified.

3.2.6 Summary

Massive search would not be possible if it was necessary to evaluate every rule. Brute's pruning strategies allow it to evaluate only a small fraction of the entire search space while guaranteeing it finds the highest-scoring rule.

Table 3.2 shows the effectiveness of Brute's pruning strategies. The table lists for each benchmark database the size of the search space, the number of rules Brute had to evaluate to find the highest-scoring rule, and the reduction in the number of rules Brute had to search due to rule pruning. The numbers for each database were computed by running Brute on ten randomly-selected samples of half the database and averaging the results. The results show that pruning can reduce the size of the search space by as much as a factor of 10^{70}. Without these large reductions in the search space, massive search would not be practical.

Table 3.2: Rule pruning avoids searching portions of the search space that can be proven not to contain the highest-scoring rule. The table shows that rule pruning can make extremely-large search spaces manageable.

Database	Search Space Size	Rules Searched	Pruning Factor
Autos	10^{60}	1,386,860	10^{54}
Cancer	10^{12}	33,255	10^{7}
Chess	10^{17}	1,946	10^{14}
Credit	10^{34}	1,684,940	10^{28}
Diabetes	10^{28}	5,762,760	10^{21}
Glass	10^{28}	44,129	10^{23}
Hepatitis	10^{24}	1,772	10^{21}
Iris	10^{10}	438	10^{7}
Lymphography	10^{17}	2,165	10^{13}
Monk1	10^{5}	262	10^{3}
Monk2	10^{5}	6,406	10^{2}
Monk3	10^{5}	374	10^{3}
Mushroom	10^{35}	692	10^{32}
Promoters	10^{74}	7,362	10^{70}
Soybean	10^{30}	8,679	10^{26}
Thyroid	10^{35}	30,183	10^{30}
Tumor	10^{9}	12,207	10^{4}
Voting	10^{7}	81	10^{5}

3.3 Rule Evaluation

The second component of Brute's execution time is the cost of evaluating each rule. Brute evaluates rules using the Laplace accuracy formula described earlier:

$$\mathcal{L}(r) = \frac{|E_{+}(r)| + k \times \mathcal{A}_{base}}{|E(r)| + k}.$$

The parameter controlling the variance of the assumed prior distribution, k, is provided by the user. The accuracy of the empty rule, $\mathcal{A}_{base}$, is computed as a preprocessing step and does not contribute significantly to the overall running time. The main cost is determining the number of positive examples $|E_{+}(r)|$ and the total number of examples $|E(r)|$ matched by the rule.

The simplest approach to rule evaluation is to scan the database and count the number of positive and negative examples covered by the current rule. A rule covers a particular example if each of the tests in a rule's antecedent hold for that example. The complexity of this approach is $O(Nl)$ where N is the number of examples in the database and l is the number of tests in a rule's antecedent.

We can improve the efficiency of rule evaluation by caching the examples matched by each rule. Since a rule's children add only one test to the rule's antecedent, the examples matched by the current rule match all but one of the tests that must be matched by its children. Passing the examples matched by the current rule to each child makes it possible to evaluate each child by selecting the subset that matches the child's additional test. Example caching reduces the cost of rule evaluation by reducing the number of examples processed at each node and by reducing the number of comparisons for processing an example from l to two. The complexity of rule evaluation using example caching is $O(M)$ where M is the number of examples matched by the parent rule.

Example caching can be further improved by splitting the database into positive instances and negative instances and maintaining this distinction as we pass results

```
PROCEDURE ComputeScore(ParentRule, NewRule, NewTest)
    NewTest.Positives = ∅
    FOREACH Example IN ParentRule.Positives DO
        IF PassesTest(NewTest, Example) THEN
            NewTest.Positives = NewTest.Positives ∪ Example
        ENDIF
    END
    NewTest.Negatives = ∅
    FOREACH Example IN ParentRule.Negatives DO
        IF PassesTest(NewTest, Example) THEN
            NewTest.Negatives = NewTest.Negatives ∪ Example
        ENDIF
    END
    PosCount = Length(NewTest.Positives)
    NegCount = Length(NewTest.Negatives)
    NewRule.Score = ScoreFunc(PosCount, NegCount)
    NewRule.MaxScore = ScoreFunc(PosCount, 0)
END
```

Figure 3.7: Algorithm for estimating rule accuracy using example caching. The algorithm stores the examples matched by each rule so its children can be evaluated by only checking the condition added at each level.

from one rule to another. This distinction eliminates the need for determining which examples match the goal predicate. Figure 3.7 shows the complete algorithm for rule evaluation using example caching.

The additional memory required for storing the example caches is minimal. In the simplest approach, examples are cached for every rule on the search stack that has yet to have its children evaluated. When learning with T tests, Brute's depth-first search algorithm can have at most $d \times T$ rules on the search stack at depth d whose children have not been evaluated. The maximum memory required for example caching is therefore $O(DTN)$ where D is the maximum search depth. When memory is scarce, the memory requirements can be reduced to $O(DN)$ by only storing the examples of the current rule's direct ancestors. If this is done, it is necessary to regenerate the

examples matched by each rule from its parent before its children can be efficiently evaluated. The extra computational cost of regenerating these examples is dwarfed by the cost of evaluating a node's children and therefore does not substantially impact running time. In fact, the examples needed to evaluate a node's children can be evaluated directly from the original database with little additional overhead and no extra memory.

It is unlikely that we can further reduce the complexity of evaluating a single rule since it is necessary to evaluate each rule against every example. We can still improve the efficiency of rule evaluation by considering the amount of CPU time it takes to process each example and attempting to evaluate multiple rules simultaneously. We discuss each of these possibilities in the next two sections.

3.3.1 Bit-Vectors

The computational costs of an algorithm are usually discussed in terms of computational complexity which ignores constant differences in running time. Large reductions in running time can often be achieved by reducing constant factors. This section presents a novel approach for evaluating rules that significantly reduces the constant factors of rule evaluation while marginally increasing the computational complexity. The improvement in processing speed offered by this new approach easily outweighs the increased computational complexity on real databases. Our approach can speed performance by as much as fifteen times and allows a search that would otherwise take a week to complete in half a day.

Our approach is to represent the examples matched by a rule using bit-vectors. A bit-vector is a string of 0's and 1's where each bit represents a single example. An example is considered part of the set if and only if the bit representing the example is set to 1.

The basic computation of rule evaluation can be thought of as computing the intersection of two sets: the set of examples matched by the parent rule and the

```
PROCEDURE ComputeScore(ParentRule, NewRule, NewTest)
    PosCount = 0
    NegCount = 0
    FOR i = 1 TO Length(NewTest.PositiveVector) DO
        NewRule.PositiveVector[i] =
                NewTest.PositiveVector[i] & ParentRule.PositiveVector[i]
        PosCount = PosCount + BitCount(NewRule.PositiveVector[i])
    END
    FOR i = 1 TO Length(NewTest.NegativeVector) DO
        NewRule.NegativeVector[i] =
                NewTest.NegativeVector[i] & ParentRule.NegativeVector[i]
        NegCount = NegCount + BitCount(NewRule.NegativeVector[i])
    END
    NewRule.Score = ScoreFunc(PosCount, NegCount)
    NewRule.MaxScore = ScoreFunc(PosCount, 0)
END
```

Figure 3.8: Algorithm for estimating rule accuracy using a bit-vectors. The unary '&' denotes bitwise-and. The length of a bit-vector is the number of *words* needed to store all examples. The BitCount() function counts the number of bits set in a word.

set of examples matched by the rule's additional conjunct. If we represent each of these sets using bit-vectors, we can compute the intersection of the sets using a bitwise-and operation. On a 32-bit machine, a bitwise-and operation processes 32 examples simultaneously. The bit-vector representation is efficient because of this internal parallelism.

Figure 3.8 shows the algorithm for rule evaluation using bit-vectors. The algorithm computes the bitwise-and for the positive and negative example vectors. The bitwise-and computation is performed by computing the bitwise-and of the individual words that make up each vector.

The bit-vector algorithm has the potential to be thirty-two times faster than simply scanning the database if the time for executing each algorithm's inner loop is comparable. The total time for rule evaluation using bit-vectors is

$$Time_b = \frac{1}{32} C_b \times N,$$

where C_b denotes the CPU time of executing the inner loop. The total time for the scanning algorithm presented earlier is

$$Time_s = C_s \times M.$$

Whether the bit-vector algorithm is an improvement over the scanning algorithm depends on the actual values of C_s, C_b, and M.

The values of C_s and C_b depend on the computer used for testing. While the absolute numbers will vary from machine to machine, the relative values of C_b and C_s should be similar across architectures. We estimate C_b and C_s for a single machine and assume the relative values of our estimates are valid across CPU types. We used an IBM PowerPC-41T workstation for our experiments, a 32-bit machine.

We computed our estimates for C_b and C_s by executing an efficient C implementation of each algorithm's inner loop ten million times and gathering timing information. The results show that $C_b \approx 331$ns and $C_s \approx 193$ns. The running time of each algorithm is therefore

$$\begin{aligned} Time_b &\approx \frac{1}{32} \times 331 \times N \text{ ns} \\ &\approx 10N \text{ ns}, \\ Time_s &\approx 193M \text{ ns}. \end{aligned}$$

When $M = N$, bit-vectors outperform scanning by a factor of 19. When $M < N$, as is normally the case, the performance improvement is reduced. The size of the reduction depends on the average size of the example sets passed to the rule evaluation algorithm. Table 3.3 shows the ratio of M/N for our benchmark databases and how these values translate into performance improvements on a PowerPC-41T workstation. The ratio of M/N for each database was computed by running Brute

Table 3.3: The speedup afforded by bit-vectors depends on the ratio of the number of examples that need to be evaluated for a rule (M) and the number of total examples (N). The graph shows this ratio for our benchmark databases along with the corresponding speedup for bit-vectors.

Database	M/N	Speedup of Bit-Vectors
Autos	0.35	6.5
Cancer	0.34	6.4
Chess	0.49	9.0
Credit	0.48	9.0
Diabetes	0.30	5.6
Glass	0.40	7.4
Hepatitis	0.73	13.7
Iris	0.60	11.2
Lymphography	0.52	9.8
Monk1	0.58	10.9
Monk2	0.26	4.9
Monk3	0.55	10.3
Mushroom	0.66	12.2
Promoters	0.70	13.1
Soybean	0.28	5.1
Thyroid	0.87	16.3
Tumor	0.24	4.5
Voting	0.68	12.7
Average	0.50	9.4

on ten randomly-selected samples of half the database and averaging the results. The average value of M across all databases is $0.50N$ resulting in a typical performance improvement of 9.4. In the worst case, M was $0.24N$ resulting in a performance improvement of 4.5. In the best case, the performance improvement was 16.3. These experiments demonstrate that bit-vectors provide substantial performance benefits.

The benefits of a bit-vectors should increase with the new processor technologies

now coming to market. The trend of 64-bit processors should double the relative performance of bit-vectors since it enables them to process twice as many examples in a single instruction. If 128-bit processors ever become popular, the gap will further widen.

Brute's bit-vector code uses 16-bit table lookups to count the number of bits set in a word because current architectures do not support a bit-count instruction. More than half the time spent in Brute's rule evaluation code is used for bit counting. Newer architectures such as the Ultra-SPARC [Weaver&Germond 94] which support a bit-count instruction should double performance.

Bit-vectors introduce some additional memory overhead. Brute requires $NT/8$ bytes of memory to store the bit-vectors for a database with N examples and T tests. Since the number of tests in a database is at most $2AV$ where A is the number of attributes and V is the average number of values per attribute, the amount of memory for bit-vectors can be rewritten as $NAV/4$ bytes. Since the amount of memory required to store the database itself is $4NA$ bytes, bit-vectors increase the memory requirements of massive search by a factor of $V/16$. This is often not a problem for the common case in which V is small, but can be problematic for databases with large numbers of values per attribute.

3.3.2 Partitioning and Sorting

An alternative for improving rule evaluation speed is to use partitioning and sorting techniques to evaluate multiple rules simultaneously. Partitioning and sorting techniques were first used in ID3 [Quinlan 86] and have recently been extended in Apriori [Agrawal *et al.* 96]. We present them here to compare them with bit-vectors. Partitioning and sorting are also useful for analyzing Brute's complexity.

Partitioning algorithms evaluate all tests for each discrete attribute using a single pass through the example set. Let A_i denote a discrete attribute which takes on five values $v_1 \ldots v_5$. The values of A_i partition the example set into five distinct sets

$\{S_1, \ldots, S_5\}$. Each set S_j contains the examples matching the test $A_i = v_j$. Therefore, computing this partition is sufficient to evaluate each test of the form $A_i = v_j$. We can compute this partition in a single pass by walking through the elements of the example set and placing each example in the appropriate partition. The complexity of evaluating all equality tests for a particular attribute is $O(M)$. This improves on the $O(VN)$ complexity of performing the same operations using bit-vectors, where V denotes the average number of values per attribute. The improvement afforded by partitioning is greatest when the number of values per attribute is large.

The efficiency of partitioning is diminished when we consider the additional cost of computing the *examples* matched by tests of the form $A_i \neq v_j$. This reduction in efficiency has not been previously noted because previous systems that have used partitioning for inequality have used them in the context of greedy search. A test $A_i \neq v_j$ will match the complement of the examples matched by $A_i = v_j$. We can count the *number* of instances matched by an inequality test by complementing the number of instances matched by the corresponding equality test. Since the number of positive and negative instances matched by a rule is all that is needed to estimate accuracy, the accuracy of inequality tests can be estimated without increasing complexity.

We must also compute the examples matched by a test if we wish to efficiently evaluate the test's children. The set of examples matched by an inequality test can be computed by scanning the example set and storing each example along with all the tests it matches. Unlike equality tests, an example will match all but one inequality test. The cost of computing the examples matched by both equality and inequality tests is therefore $O(VM)$ — about the same as using bit-vectors. However, much of this additional complexity can be hidden within the search space. Each test for which we generate examples has children that also must be evaluated. If we move the generation of examples to the first step of evaluating the children, we can hide the complexity of example generation in the cost of evaluating the children. The cost of evaluating a child *without* generating examples is $O(M)$. The cost of generating

examples at a child node is $O(M')$ where M' denotes the number of examples matched by a rule's grandparent. The total complexity for partitioning is therefore $O(M')$.

While partitioning reduces the complexity of rule evaluation, the execution constants determine which algorithm will perform better in practice. Which algorithm will perform better depends on the algorithm's execution constants. Since the constant factor for bit-vectors is 10 ns, the total time for evaluating all V tests using bit-vectors is

$$Time_b \approx 10\,VN \text{ ns.}$$

By executing the inner loop of the partitioning algorithm ten million times, we estimated the constant factor for partitioning to be about 283 ns,

$$Time_p \approx 283M' \text{ ns.}$$

Since M' is typically about one-half N, partitioning will perform better than sorting whenever $V > 14$. This occurs in only 1 of 279 discrete attributes in our database sample. The advantage of bit-vectors is greatest for binary attributes ($V = 2$) where it outperforms partitioning by a factor of seven.

While partitioning cannot be applied to numerical attributes, similar results can be achieved by sorting [Quinlan 86]. With a sorted example set, we can estimate the accuracy for all tests by scanning the sorted array and counting the number of positive and negative examples that appear before each unique value v. These counts can be used to compute the score of the rules formed with $A_i \leq v$ and $A_i > v$.

The complexity of evaluating all tests for a numerical attribute A_i using a standard comparison sort is $O(M \log M)$. We can improve the complexity to $O(N)$ by using a counting sort [Cormen *et al.* 90]. Counting sorts are almost always faster than a comparison sort for rule evaluation because M is usually a large fraction of N.

The cost of evaluating a numerical attribute using bit-vectors is $O(CN)$, where C is the number of numerical tests that must be evaluated. A counting sort therefore

improves complexity by a factor of C. Whether this improvement in complexity reduces running time depends on the relative speeds of each algorithm's inner loop. The total time to evaluate all tests on a numerical attribute using bit-vectors is

$$Time_b \approx 10CN \text{ ns.}$$

We executed counting sort's inner loop ten million times and computed the time of the inner loop to be about 1,332 ns. The total running time is therefore

$$Time_s \approx 1332N \text{ ns.}$$

Sorting is the better choice when $C > 132$.

Some numerical attributes, such as a person's age or height, can only take on a finite range of values. When a numerical attribute's range is less than 132 values, bit-vectors are the better choice. On some numerical attributes, each example tends to have its own value. The value of C for these attributes grows with the size of the database. However, the average value of C is often small due to rule pruning and the structure of the search space. As a result, the number of unique values for a numerical attribute must be significantly higher than 132 before sorting is beneficial. The exact change-over point is difficult to estimate and is left for future work. None of the benchmark databases experimented with in this thesis had an average value of C greater than 132.

Both partitioning and sorting require the examples matched by a rule to be stored as example lists. Neither of these techniques can be efficiently applied to bit-vectors. Using partitioning for discrete attributes requires the use of either scanning or sorting for numerical attributes. Similarly, using sorting for numerical attributes requires the use of either scanning or partitioning for discrete attributes. While using a combination of partitioning and sorting appears ideal, the use of sorting in conjunction with partitioning actually makes partitioning less efficient. The problem is that the order

Table 3.4: Comparison of the various methods of rule evaluation. The table shows that the method with the least computational complexity, partitioning and sorting, has the highest constant factors. Bit-vectors perform best in practice because of their low constant factors and because C and V are often small.

Algorithm	Running Time for Discrete Attributes	Running Time for Numerical Attributes
Scanning	$193\,VM$ ns	$193\,CM$ ns
Bit-Vectors	$10\,VN$ ns	$10\,CN$ ns
Partitioning and Scanning	$283M'$ ns	$193\,CM$ ns
Partitioning and Sorting	$1209M'$ ns	$2500N$ ns

of the example lists affects CPU cache performance. A partitioning algorithm processing an ordered example list reads the database linearly. However, sorting rearranges the example lists. Scanning the database non-linearly results in poor locality and poor cache performance. The poor cache performance of partitioning when combined with sorting reduces the effectiveness of partitioning by a factor of five. Combining scanning with sorting is no better since scanning algorithms have similar problems with locality.

The choice of rule-evaluation algorithm is complex. Table 3.4 shows the time complexity of each option. Simple scanning has the worst performance. Partitioning and scanning perform well on databases with a large number of values per discrete attribute. Partitioning and sorting is best for domains with many numerical attributes that take on thousands of values. Bit-vectors on the other hand are extremely fast on databases with both discrete and numerical attributes as long as the number of values per attribute, be it discrete or numerical, is not too extreme. Brute uses bit-vectors because the databases it encounters usually meet these criteria.

3.4 Complexity

The complexity of Brute is useful for understanding how Brute's performance is affected by various parameters. The complexity of Brute is usually not useful for estimating running times because worst-case complexity measures do not take into account pruning. While an average-case analysis of pruning performance would be useful, it is unclear what distribution of learning tasks should be used to base such an analysis. An analysis on randomly-distributed problems is not helpful since it is the structure of each database that makes learning difficult.

We analyze Brute's complexity in three stages. We first consider the complexity of Brute with only equality tests. We then consider the added cost of inequality tests before finally considering numerical attributes. The analysis is initially done using partitioning and sorting because these are the most efficient. We later discuss the added complexity of bit-vectors.

3.4.1 Equality Tests

The size of Brute's search space with equality tests depends on the number of discrete attributes A and the number of values per attribute V. There are $V+1$ possibilities for each attribute in any given rule: either no test on the attribute appears or a test appears with one of its V different values. Since there are $V+1$ possibilities for each of the A attributes, the total number of possible rules is $(V+1)^A$.

In the worst case, Brute must evaluate every rule in its search space. Each rule must be compared with a maximum of N examples. While this suggests Brute's running time is $O((V+1)^A N)$, a better bound can be achieved by considering partitioning.

The effect of partitioning is best explained using a combination of counting and complexity arguments without reference to Brute's actual search algorithm. An abstract rule is an if-then rule which lists the attributes compared in each test with-

out specifying the values they are compared against. An example abstract rule is "*If* $A = ? \wedge B = ?$ *then class = pos.*" If both attributes A and B are binary attributes, then this abstract rule represents four concrete rules:

$$\begin{array}{l} \textit{If } A = \textit{true} \wedge B = \textit{true then class} = \textit{pos} \\ \textit{If } A = \textit{true} \wedge B = \textit{false then class} = \textit{pos} \\ \textit{If } A = \textit{false} \wedge B = \textit{true then class} = \textit{pos} \\ \textit{If } A = \textit{false} \wedge B = \textit{false then class} = \textit{pos}. \end{array}$$

There are 2^A abstract rules. Abstract rules are interesting because partitioning can be used to evaluate all concretizations of an abstract rule in one pass. Partitioning works because each combination of values for the target attributes maps to a unique concretization. Since the cost of evaluating all concretizations of an abstract rule is $O(N)$, the complexity for evaluating all concretizations of all abstract rules is

$$O(2^A N).$$

While the nodes in Brute's search space are concrete rather than abstract, the computational cost of evaluating all concrete rules is the same as evaluating all abstract rules. Each abstract rule represents several rules in Brute's search space, and each rule in Brute's search space is represented by exactly one abstract rule. Since the cost of evaluating a single abstract rule is the same as evaluating each of its concretizations, the complexity of the two approaches is the same, $O(2^A N)$. This is a factor of $(V-1)^A$ better than the previously computed bound.

The complexity of learning with equality tests does not depend on the number of values per attribute. A database can have an extremely-large number of values per attribute without affecting Brute's running time on equality tests.

3.4.2 Inequality Tests

The cost of massive search quickly increases when we allow inequality tests. The cost of evaluating all concretizations of an abstract rule with inequalities is not linear

in the number of examples but exponential in the number of values per attribute. Abstract rules do not help derive a better complexity estimate for inequality tests.

There are 2^V possible sets of inequalities for each attribute. The total cost of evaluating all inequality tests is $O((2^V)^A M) = O(2^{VA} M)$. The complexity of evaluating both equality and inequality tests is $O((2^A)(2^{VA})M) = O(2^{(V+1)A} M) \approx O(2^{VA} M)$, the same as with only inequality tests.

The complexity of $O(2^{VA} M)$ is significantly worse than the complexity of $O(2^V N)$ without inequality tests. The complexity of inequality tests grows exponentially with the number of values per attribute. The amount of search Brute can perform therefore is limited when there is a large number of values per attribute and inequality tests are needed.

3.4.3 Numerical Attributes

Let V denote the average number of values per numerical attribute. For each numerical attribute, there are a maximum $V - 1$ possible thresholds that can be used for either $\leq$ and $>$ tests. As a result, there are V possible values for an attribute in any given rule: the attribute either does not appear or it appears with one of $V - 1$ values. The total number of rules is therefore $O(V^{2A})$. The total complexity using scanning or bit-vectors is $O(V^{2A} N)$. Sorting reduces the cost of rule evaluation from $O(VN)$ to $O(N)$, a factor of V improvement. The factor of V improvement does not significantly change search complexity since

$$O((V^{2A} N)/V) = O(V^{2A-1} N) \approx O(V^{2A} N).$$

While sorting does not significantly improve complexity, a factor of V improvement in running time is important in practice.

3.4.4 Effect of Depth Bound

Brute uses a depth bound when its search space is too large to search in the time available. The depth bound affects the complexity of equality, inequality, and numerical tests differently. This in turn changes the relative costs of each type of attribute.

Abstract rules are useful in understanding the complexity of equality tests. There are $\binom{A}{D}$ different abstract rules of length D. When using a depth bound of D, the number of abstract rules Brute considers is $\sum_{i=0}^{D}\binom{A}{i}$. Since the cost of evaluating all concretizations of an abstract rule is $O(N)$, the total cost of evaluating equality tests with a depth bound D is

$$O\left(\sum_{i=0}^{D}\binom{A}{i}N\right) \approx O\left(\binom{A}{D}ND\right).^{3}$$

The complexity of inequality tests can be understood in a similar way. First we choose D attributes, and then we select one of V values for each of the D attributes. This selection process implies the cost of evaluating inequality tests is

$$O\left(\sum_{i=0}^{D}\binom{A}{i}V^{i}M\right) \approx O\left(\binom{A}{D}V^{D}MD\right).$$

The significance of these two formulas is that the complexity increase over equality tests is only polynomial in V, where for unbounded search it was exponential in V. Inequality tests normally increase complexity exponentially because they increase the maximum rule length by a factor of V. With a depth bound in place, the use of inequality tests does not increase rule length and therefore does not impact performance as severely.

This phenomena is particularly important when we take into account rule pruning. Each test added to a rule tends to reduce the number of examples it matches by some appreciable fraction. As a result, the number of examples matched by a rule typically decreases exponentially with rule length. Because of this exponential decrease in rule

---

[3] This approximation only holds when $D < A/2$, as is frequently the case.

coverage, the average depth of the search tree tends to be logarithmic in the number of examples. This introduces an implicit depth bound which causes exhaustive search performance to behave similarly to depth-bounded search. Rule pruning causes the additional cost for inequality tests to grow polynomially rather than exponentially in practice.

The cost of numerical tests, by an identical argument, is syntactically equivalent to that of inequality tests

$$O\left(\sum_{i=0}^{D}\binom{A}{i}V^{D}M\right) \approx O\left(\binom{A}{D}V^{D}MD\right).$$

Their costs differ only because the number of values per attribute tend to be substantially larger for numerical attributes.

3.4.5 Total Complexity

Most databases contain a mixture of numerical and discrete attributes. The search-space size for a mixed database is the product of the search-space sizes for each attribute class. All rules in the combined search space can be formed by concatenating a rule containing only numerical attributes with a rule containing only discrete attributes.

Table 3.5 summarizes Brute's complexity and gives the complexity for mixed-type databases. The table uses A_n and A_d to denote the set of numerical and discrete attributes respectively. The terms V_n and V_d are used to refer to the average number of values per each numerical and discrete attribute. The second column shows Brute's complexity with a depth bound of D. The complexities are given for partitioning and sorting. The use of bit-vectors increases complexity by a factor of $O(CN/M)$, where C is the average number of tests that must be evaluated per attribute.

In all cases, Brute's complexity grows linearly with the number of examples and exponentially with the number of attributes. The number of values per attribute increases complexity exponentially in V_d when using inequality tests and exponentially

Table 3.5: Summary of Brute's complexity using partitioning and sorting. Bit-vectors increase the complexity by a factor of CN/M.

Attribute Type	Full Search	Search to Depth D
Equality	$2^{A_d}N$	$\binom{A_d}{D}ND$
Inequality	$2^{V_d A_d}M$	$\binom{A_d}{D}V_d^D MD$
Numerical	$V_n^{2A_n}M$	$\binom{A_n}{D}V_n^D MD$
Total	$2^{V_d A_d}V_n^{2A_n}N$	$\sum_{i+j=D}\binom{A}{i,j}V_d^i V_n^j MD$

in $\log V_n$ when using numerical attributes. The effect of the number of values per attribute reduces to polynomial whenever a depth bound is used.

The complexities of Table 3.5 are useful in understanding how the size of Brute's input database affects performance. The complexities are not useful in predicting total running time since rule pruning results in only a small fraction of the space being searched. The next section investigates Brute's running time on actual databases.

3.5 *Performance Analysis*

Table 3.6 shows the running time for Brute on our benchmark databases. Brute's running time was computed by taking ten random samples from each database where each sample contained half the data. Brute was run on each sample individually. The running time reported for each database is Brute's average running time on the ten samples.

Brute can completely search ten of the fourteen databases in less than one sec-

Table 3.6: Brute's running time for a complete search when evaluating rules using Laplace accuracy. Times shown are for a PowerPC-41T workstation.

Database	CPU Time (seconds)
Autos	29.6
Cancer	0.7
Chess	1.2
Credit	29.2
Diabetes	100.6
Glass	1.7
Hepatitis	0.2
Iris	0.2
Lymphography	0.2

Database	CPU Time (seconds)
Monk1	0.1
Monk2	0.3
Monk3	0.1
Mushroom	4.3
Promoters	0.3
Soybean	1.6
Thyroid	9.0
Tumor	0.5
Voting	0.2

ond. The most expensive database to search, Diabetes, takes less than two minutes. Massive search is clearly practical for these databases.

The databases in our sample, while commonly used to evaluate machine-learning algorithms, are too small to challenge Brute's search algorithm. A better understanding of Brute's performance is obtained by applying Brute to larger databases. Table 3.7 shows the ten largest databases from the UCI repository and two large Boeing manufacturing databases. Three databases, Mushroom, Chess and Thyroid, were used in the previous experiments. These databases are sufficiently large such that Brute cannot completely search three of them within 24 hours. For the databases that cannot be completely searched, the largest depth bound that can be completed within 24 hours is used. Brute could not search the Adult database because the test machine did not have the memory resources to store the 350 MB's of bit-vectors required to process this database.

Table 3.8 shows Brute's performance on these large databases. The results show

Table 3.7: Large databases for testing Brute's limits. The table contains the ten largest databases from the UCI data repository and two large Boeing databases.

Database	Examples	Classes	Discrete Attributes	V_d	Numerical Attributes	V_c
Connect-4	67,557	3	43	3	—	—
Shuttle	58,000	7	—	—	9	123
Adult	48,845	2	8	12	6	4,822
Letters	20,000	26	—	—	16	16
Mushroom	8,416	2	22	5	—	—
ANN Thyroid	7,200	3	—	—	21	69
Satellite	6,435	6	—	—	36	78
Abalone	4,177	28	1	3	7	864
Chess	3,196	2	36	2	—	—
Thyroid	3,163	2	18	2	7	165
Occupancy Time†	1,075	4	4	5	43	5
Failed Parts†	519	2	1,631	2	20	63

†Boeing manufacturing database.

Table 3.8: Brute's performance on large databases. While Brute performed well on most databases, the test machine did not have the resources to store the 350 MB's of bit-vectors required to process the Adult database.

Database	Search Depth	Size of Search Space	Rules Searched	CPU Time (hr:min)	Error Rate Reduction
Connect-4	4	10^8	10^6	3:13	2.9
Shuttle	full	10^{32}	10^3	0:08	24.3
Adult	*not enough memory*				
Letters	full	10^{35}	10^6	0:16	11.7
Mushroom	full	10^{35}	10^6	< 0:01	4.3
ANN Thyroid	full	10^{36}	10^3	< 0:01	8.1
Satellite	3	10^{10}	10^7	6:34	2.4
Abalone	full	10^{34}	10^9	18:03	3.2
Chess	full	10^{17}	10^3	< 0:01	8.0
Thyroid	full	10^{35}	10^4	< 0:01	1.8
Occupancy Time†	full	10^{53}	10^6	0:03	5.1
Failed Parts†	full	10^{13}	10^5	< 0:01	8.1

†Boeing manufacturing database.

that Brute can efficiently search many of these large databases. When Brute cannot perform a complete search, its massive, but partial, search finds rules with significantly higher estimated accuracy than those found by greedy algorithms. These improvements in estimated accuracy translate into a factor of 1.8 to 24.3 increase in the estimated error of the learned rules. While Brute's running time for a few databases is several hours, in critical applications, the large error-rate reduction is worth the extra computational costs.

3.6 Related Work

ITRule [Smyth&Goodman 91] was one of the first algorithms to apply massive search to rule learning. ITRule uses best-first search with branch-and-bound pruning to find the rule that maximizes an information-theoretic evaluation function. Brute improves on ITRule by making massive search more efficient.

Apriori [Agrawal *et al.* 96] is another recent massive-search algorithm for learning association rules. Association rules are an extension of if-then rules that allow consequents to be arbitrary conjunctions. Apriori's algorithm first finds all rules with a minimum data coverage and then uses the search results to find association rules. Apriori's search space is identical to Brute's with an evaluation function that ranks rules according to data coverage.

Apriori uses a radically different model for rule evaluation in order to support very-large databases that cannot fit in memory and must reside on external storage. Since it is expensive to read a database from external storage, it is important that the database be read as few times as possible. Apriori reduces the number of passes over the database by maximizing the number of rules evaluated in each pass. Apriori explores the search tree level by level. The entire search space for the current level is stored in memory. As the database is read, all the children for the next level are evaluated simultaneously. The database is read only once for each level in the search

space. Since most search trees are fairly shallow, the number of passes through the database is minimal. The disadvantage of this approach is that it requires enough memory to store all rules visited. This is impractical for large search spaces.

Brute's subsumption pruning and dynamic reorganization were adapted from OPUS [Webb 95], a massive-search algorithm designed for rule learning. While both algorithms use similar branch-and-bound pruning techniques, they were developed independently. Brute extends OPUS to use depth-first search and to handle numerical attributes. Brute improves the efficiency of subsumption pruning by recognizing that rule subsumption can be syntactically checked if the two rules only differ in their numerical thresholds.

Schlimmer [1993] presents a massive-search algorithm for learning determinations that uses a unique method for ensuring each pruning operation removes as many determinations as possible. The algorithm is easily adapted for rule learning. Schlimmer's algorithm maintains a table to track which rules should be pruned. The table contains one entry for each node in the search space. Whenever a rule is pruned, all specializations of the rule are marked as prunable in the table. The table is used to check if a rule should be pruned before evaluation. Pruning tables guarantee all specializations of a pruned rule are never evaluated. Brute cannot make this guarantee because all specializations of a pruned rule are not necessarily in the rule's subtree.

The fundamental limitation of Schlimmer's approach is its memory requirements. The algorithm requires one bit of memory for each rule in the search space. For a large search space such as the 10^{74} rules needed to completely search the Promoters database, Schlimmer's approach would require 10^{67} MB's to store the pruning table. With memory capacities doubling every eighteen months, this amount of memory should not be available for about 300 years.

Apriori uses a similar technique for ensuring that all specializations of a pruned rule are never evaluated. Rather than maintaining a database of all prunable rules, Apriori's database only stores the root of each pruned subtree. This database is

maintained using nested hash tables. Checking if a rule has been pruned is performed by determining if any of its generalizations appear in the nested hash table.

Nested hash tables, while requiring less memory than pruning tables, are still memory intensive. Their memory requirements grow linearly with the number of rules visited. At 50,000 rules per second and 8 bytes per rule, the memory requirements for nested hash tables can grow by 24 MB's per minute.

The potential benefits for memorizing pruned rules do not justify their high computational costs and large memory requirements. Pruning histories only eliminate the need to evaluate a rule that is a specialization of a previously pruned rule. The subtree below the specialization would be pruned anyway because rule evaluation detects all prunable subtrees. The benefits of pruning histories are therefore limited to reducing the number of rules evaluated at each node. The maximum performance improvement afforded by pruning histories is only a factor of $|\mathcal{T}|$.

3.7 Summary

Brute conducts a heuristically-guided, depth-first search of the space of conjunctive rules to find the rule that maximizes a user-supplied evaluation function. Brute uses several pruning strategies and a bit-vector representation to explore as many rules as possible. Brute can efficiently perform a complete search on our benchmark databases and can perform massive, but partial, searches on the largest databases available to the author. Whether Brute's search was partial or complete, it always found rules that better maximized the user-supplied evaluation function.

The goal of rule learning is to find a rule that best maximizes the user's *utility* function. The utility of a learned rule depends on how well the rule learned describes the real world. It is an open question as to whether Brute's ability to find higher-scoring rules corresponds to an ability to find rules with higher utility. This is the topic of the next chapter.

Chapter 4

INDUCTIVE PERFORMANCE

Chapter 3 showed that massive search is practical and learns rules with substantially-higher scores than greedy search. The goal of learning is not finding higher-scoring rules but finding rules that better characterize the real world described by the database. We therefore measure Brute's success by how well the rules learned describe the real world.

Brute's ability to find higher-scoring rules should result in learning more accurate rules if its evaluation function is a good indicator of rule performance. However, experimental results from Webb [1993] and Quinlan and Cameron-Jones [1995] suggest that massive search actually reduces inductive performance. Quinlan and Cameron-Jones call this phenomenon *oversearching* and argue it is a direct consequence of large-scale search. This chapter analyzes oversearching and shows that oversearching can be partially explained by overfitting. When a bias for short rules is added to the evaluation function, oversearching is significantly reduced. Our analysis suggests that oversearching may not be inherent in large-scale search but depends on the evaluation function employed.

The next section presents Quinlan and Cameron-Jones' evidence for oversearching and discusses their argument that oversearching is an unavoidable consequence of massive search. Section 4.2 shows how Quinlan and Cameron-Jones' results can be partially attributed to overfitting and presents a new evaluation function with better performance. Section 4.3 discusses related work before concluding in Section 4.4.

4.1 Oversearching

The true test of massive search is whether it learns rules that better model the real world than greedy search. Webb [1993] analyzed the inductive performance of massive search and found that additional search does not always offer an improvement. Quinlan and Cameron-Jones [1995] conducted a detailed analysis of massive search and found similar results. This section reproduces their experiments and discusses possible explanations for massive search's poor performance.

Quinlan and Cameron-Jones evaluated the effects of increased search by searching for the rule that maximizes Laplace accuracy using a beam search with beam widths varying exponentially from 1 to 512. The larger the beam width, the more extensive the search. They randomly chose half the data for training and half the data for evaluating the rules learned. For each database, learning was repeated 5,000 times.[1] Their experimental methodology is summarized below:

```
Repeat 5,000 times:
    Split the data randomly into equally-sized training and test data.
    For beam widths w = 1, 2, 4, ..., 512:
        For each goal class:
            Identify the rule with the highest Laplace accuracy
                during a beam search of width w.
            Determine the rule's accuracy on the test data.
```

Figure 4.1 shows the results of repeating this experiment on our sample databases. The accuracies shown are weighted averages of the test-data accuracy of the rules learned for each class, weighted by the relative frequency of each class in the training data. The results are averaged over the 5,000 iterations. The error bars show the 99% confidence interval for each measurement.

[1] Learning was repeated 25,000 times for the Diabetes and Voting databases to ensure statistically-meaningful results.

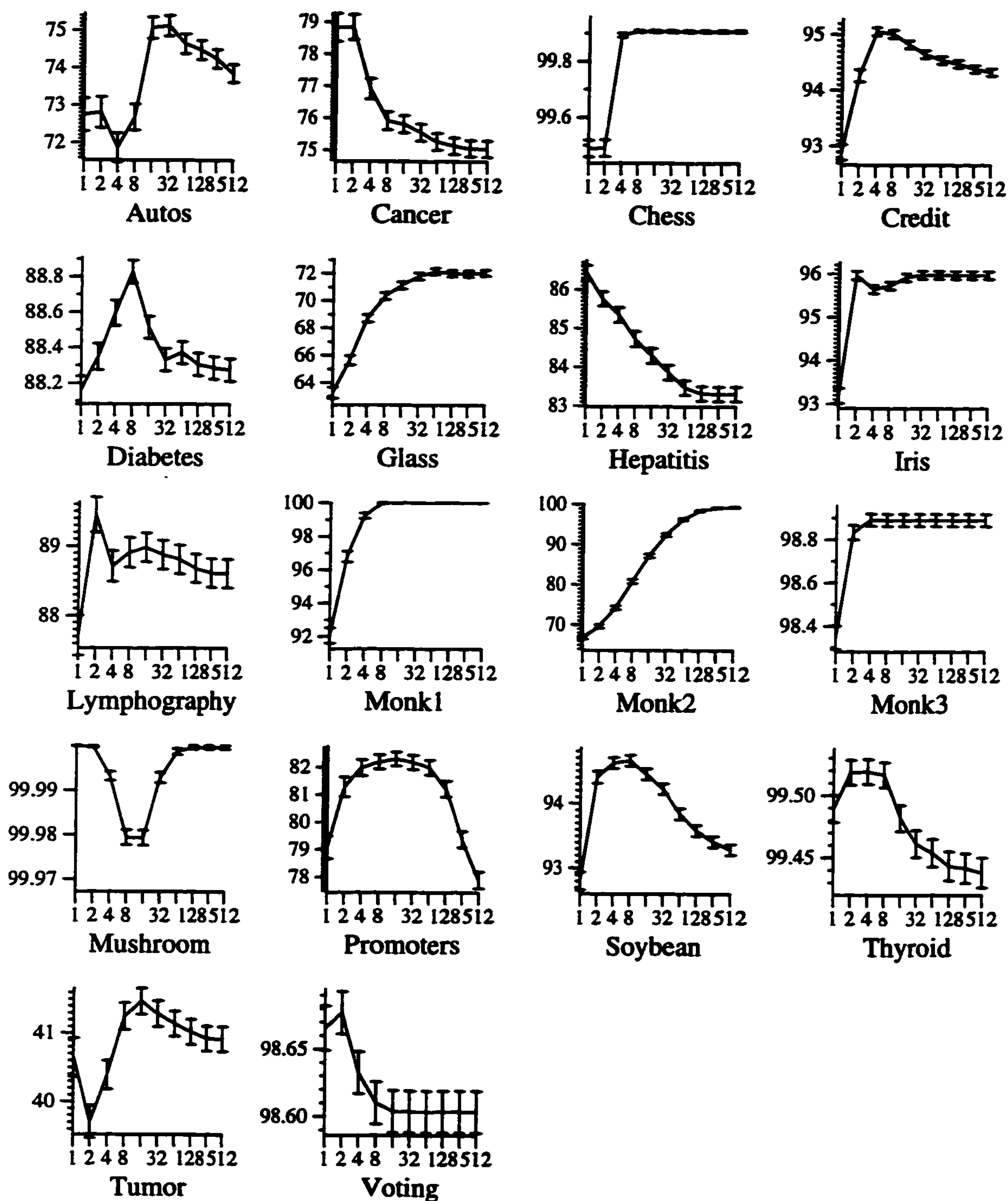


Figure 4.1: Reproduction of Quinlan and Cameron-Jones' experiments showing reduced predictive ability with additional search. The graphs show test accuracy for rules learned with beam widths growing exponentially from 1 to 512.

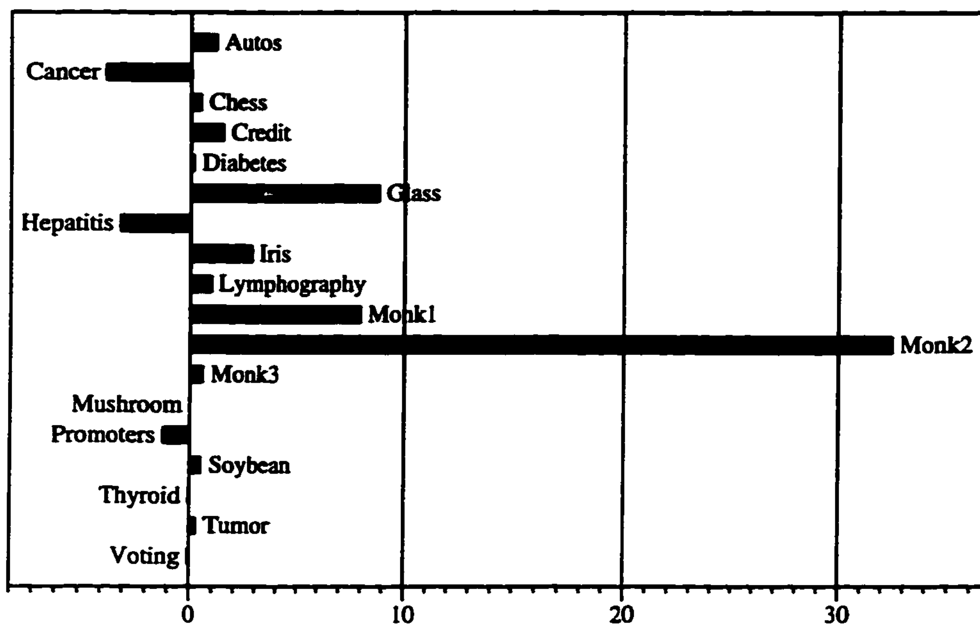


Figure 4.2: Difference in test accuracy between the rules learned using beam-512 search and those learned using greedy search. The graph shows that beam-512 search outperforms greedy search on most databases and often results in large increases in accuracy.

The experiments show that massive search typically learns better rules than greedy search. Greedy search is equivalent to a beam search with a beam width of one. A beam width of 512 approximates massive search. Figure 4.2 highlights the accuracy improvement afforded by massive search. Massive search performs better on twelve databases, while greedy search performs better on only five. The margins of improvement for massive search are much larger than those for greedy search. While greedy search offered a maximum improvement of 3.8 percentage points, massive search learns rules that are as much as 32 percentage points higher.

While massive search outperforms greedy search, the graphs demonstrate an over-searching effect. Ideally, the test accuracy should increase monotonically with beam size since the rules learned with larger beams have higher scores. Only the graphs

for the Chess, Glass, and the three Monks databases exhibit this ideal behavior. The remaining graphs show varying degrees of oversearching. The Cancer and Hepatitis databases highlight the extreme case of accuracy strictly decreasing with additional search. Many of the graphs are hill shaped where accuracy first increases but later decreases.

Quinlan and Cameron-Jones argue that oversearching is a direct result of additional search. Their argument is based on the existence of *fluke rules*, rules that score well on the training data but perform poorly on the test data. The more rules an algorithm visits, the greater chance of encountering a fluke rule with poor test accuracy. They conclude that the number of rules searched should be limited in order to learn rules with high test accuracy.

Quinlan and Cameron-Jones [1995] propose limiting the number of rules searched based on a probabilistic argument similar to that used by Blumer *et al.* [1987] to detect overfitting. The algorithm they developed, *layered search*, performs separate searches with beam widths exponentially growing from 1 to 512. Layered search uses its probabilistic theory to decide which of its several searches is likely to have found the best rule and returns the rule selected by that search. Figure 4.3 shows the performance of layered search superimposed on the graphs of Figure 4.1. The performance of layered search is indicated by boxes that are placed at the intersection of layered search's test accuracy and the average beam width it selects. Layered search frequently does well in choosing how much search to perform. On the Credit database, the average beam width selected is close to the maximum of the curve. Figure 4.4 compares massive and layered search and shows that layered search outperforms massive search on twelve of the eighteen databases. Massive search performed better on only three databases. However, massive search excels on the Monk2 database where it finds rules whose accuracy averages fifteen percentage points higher. The Monk2 database is an artificial domain in which the best rules test all six attributes. Greedy and layered search's bias for short rules makes it difficult to learn the long

rules required to perform well in this domain.

While layered search performs quite well, there is an alternative explanation for its success. A good evaluation function should be highly correlated with test-data performance. If the highest-scoring rules are not performing well on the test data, the evaluation function must not be sufficiently correlated with test-data performance. A better evaluation function may help realize the potential benefits of massive search. The next section investigates Laplace accuracy and discusses how its performance may be improved.

4.2 Overfitting

Oversearching can be explained by a poor correlation between Laplace accuracy and test data performance. A poor correlation implies Laplace accuracy is either failing to make the correct tradeoff between data accuracy and data coverage or is failing to take into account additional parameters that affect test data performance. This section analyzes the rules learned by massive and layered search to better understand why rules with lower Laplace accuracy often perform better than higher-scoring rules. Our approach will be to analyze single learning episodes to gain insight into why Laplace accuracy performs poorly.

Table 4.1 compares the rules learned by layered and massive search for a single run on the Promoters database. The table shows the accuracy and coverage of each rule on both the training and the test data. Both rules have 100% accuracy on the training data, but the rule learned by massive search covers 30% more examples. This pattern is typical. The data accuracy of the rules learned by both algorithms averaged 99.8%. Since almost every rule learned is 100% accurate, the rules learned by massive search differ mainly by their increased coverage.

For two rules with identical data accuracy, the rule that covers the most examples is expected to have the highest accuracy on the test data. However, Table 4.1 shows

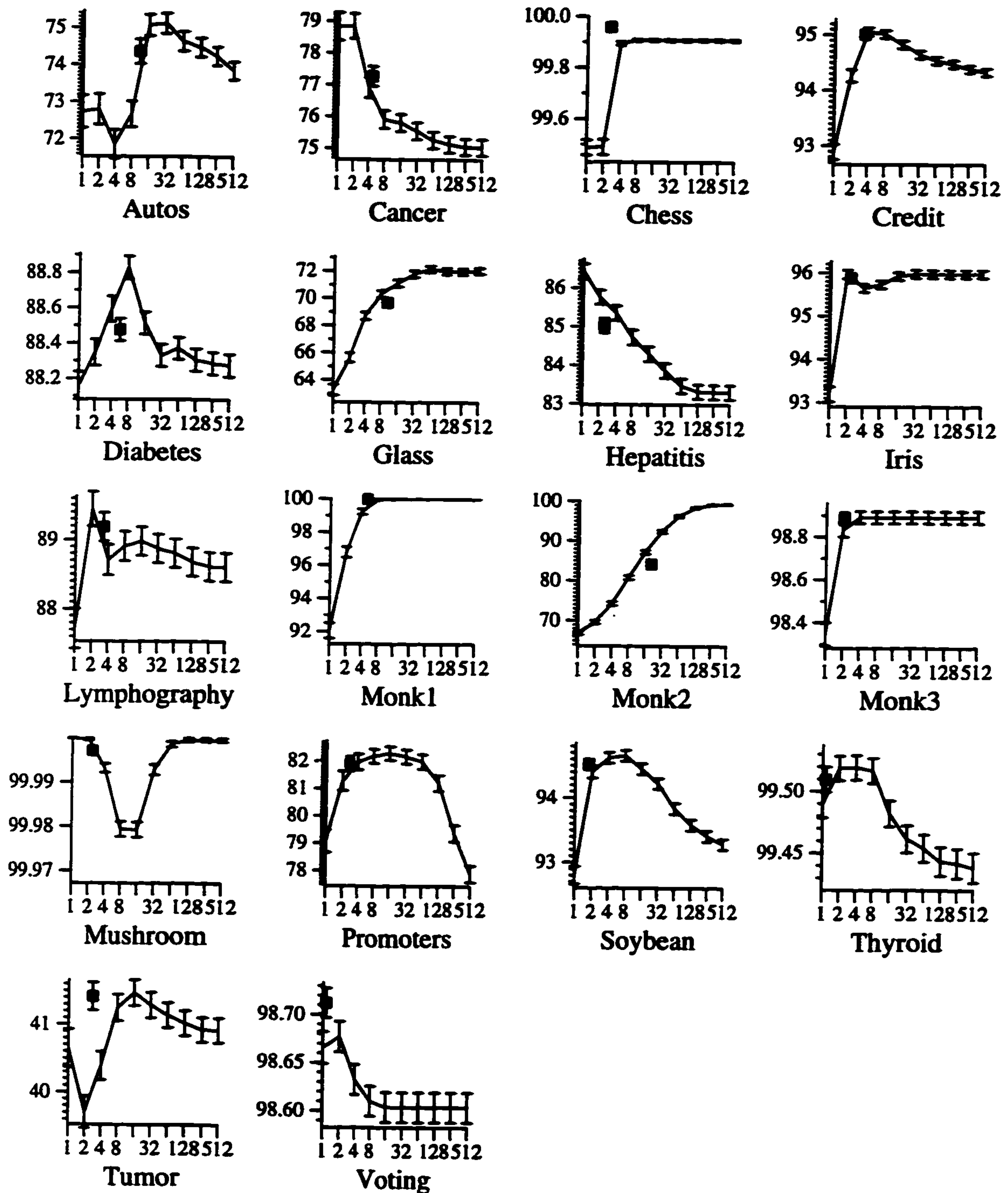


Figure 4.3: Inductive performance of layered search superimposed on the graphs of Figure 4.1. Each box shows the average accuracy of layered search and the beam width it uses to achieve that accuracy. Layered search performs well and often selects beam widths near the top of each curve.

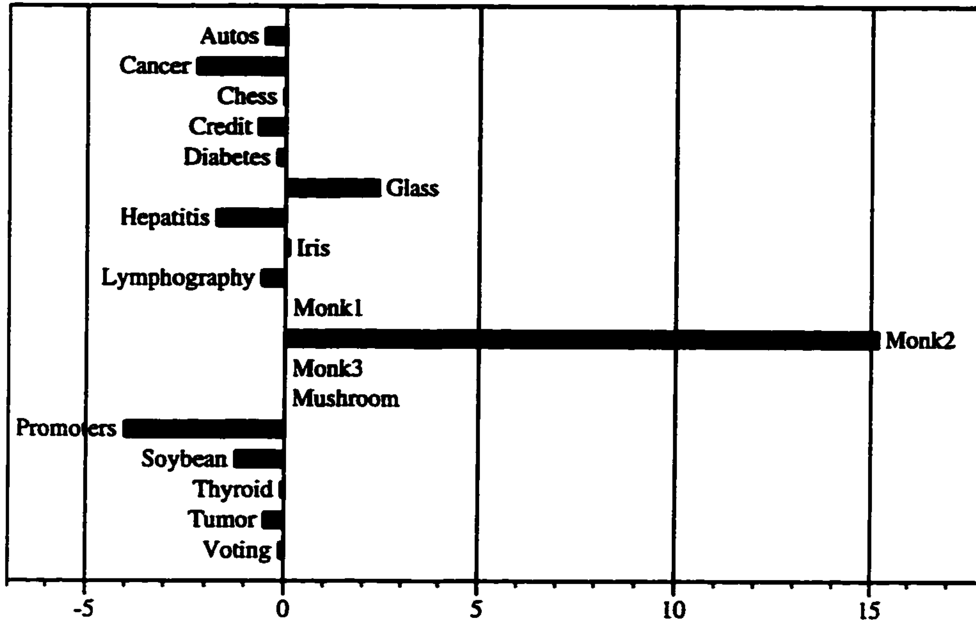


Figure 4.4: Comparison of the inductive performance of beam-512 search and layered search showing that layered search typically performs better.

that the massive-search rule performs substantially worse than the layered-search rule even though the massive-search rule has better training-data coverage. There must be additional factors that influence a rule's performance that are not captured by accuracy and coverage. One possible factor is rule length. The rule learned by massive search is considerably longer than the rule learned by layered search. This pattern holds for many of the rules learned by massive search. Massive search learned rules that are on average 29% longer than those learned using layered search. Layered search's bias for short rules may explain why it performs so well.

A preference for short rules is often used to avoid the problem of overfitting. Overfitting is similar to oversearching. Overfitting occurs when an algorithm learns complex rules that too closely mimic the available data. The classic example of overfitting is a conjunctive rule that contains enough conjuncts to exactly specify a

Table 4.1: Example rules learned using massive and layered search. While the massive-search rule is expected to have better inductive performance since it covers more examples, it actually performs worse.

Rule	Training Data		Test Data	
	Accuracy	Coverage	Accuracy	Coverage
Massive Search *If* $N9 \neq c \wedge N26 \neq t \wedge N47 \neq c \wedge N48 \neq g \wedge N56 \neq g \wedge N59 \neq c$ *then* $class = neg$	100.0	78.3	33.0	10.0
Layered Search *If* $N10 \neq t \wedge N17 = a$ *then* $class = neg$	100.0	60.9	100.0	20.0

single example. As long as the database is consistent, multiple rules of this form can be created for each training example. However, since these rules are always present, they give little information about the patterns present within the training data. As a result, these complex rules often perform poorly on future data. A preference for short rules tends to reduce overfitting by discouraging the learning of overly-complex rules. If the oversearching problem is caused by overfitting, layered search's preference for short rules would bolster its performance.

Overfitting typically occurs when a complex rule matches only a few examples, but the rules learned by massive search have high coverage. If overfitting is occurring, it must be very different from the low-coverage overfitting described earlier. Can overfitting occur on rules with high coverage? Absolutely.

High-coverage overfitting can be explained in terms of low-coverage overfitting. The antecedent of the rule learned by massive search in Table 4.1 can be rewritten using DeMorgan's law as

$$\overline{N9 = c \vee N26 = t \vee N47 = c \vee N48 = g \vee N56 = g \vee N59 = c},$$

Table 4.2: Analysis of the individual conjuncts forming the massive-search rule from Table 4.1. The table shows the accuracy and coverage of the negation of each conjunct for predicting the negation of the goal predicate on both the training and test data. The high accuracy but low coverage of each negated conjunct results in poor test-data performance.

Rule	Training Data		Test Data	
	Accuracy	Coverage	Accuracy	Coverage
If $N9 = c$ *then class* $\neq$ *neg*	62.5	16.7	30.8	17.4
If $N26 = t$ *then class* $\neq$ *neg*	82.4	46.7	33.3	21.7
If $N47 = c$ *then class* $\neq$ *neg*	83.3	16.7	55.6	21.7
If $N48 = g$ *then class* $\neq$ *neg*	71.4	16.7	35.3	26.1
If $N56 = g$ *then class* $\neq$ *neg*	88.9	26.7	53.8	30.4
If $N59 = c$ *then class* $\neq$ *neg*	85.7	20.0	20.0	8.7

where the over-bar indicates negation. The examples matched by the rule's antecedent is therefore all examples *not* matched by the disjunction

$$N9 = c \vee N26 = t \vee N47 = c \vee N48 = g \vee N56 = g \vee N59 = c. \tag{4.1}$$

For the massive-search rule to have high accuracy, this disjunction must cover as many negative examples as possible. If this disjunction matches all negative examples, the rule would be 100% accurate.

The number of negative examples covered by disjunction (4.1) depends on the ability of each disjunct to predict negative examples. Table 4.2 shows the performance of each disjunct for predicting negative examples on both the training and test data. Each of these disjuncts has high accuracy and low coverage on the training data. Their low coverage suggests that they will not perform well on the test data. Table 4.2 shows that this prediction of poor performance is correct. The poor predictive ability of each disjunct causes the rule learned by massive search to perform poorly.

High-coverage overfitting occurs with long rules because many terms are needed

for several low-coverage disjuncts to cover most negative examples. High-coverage overfitting can be reduced using the same preference for short rules used to avoid low-coverage overfitting.

Preferences for short rules are often encoded using the Minimum Description Length (MDL) principle [Rissanen 78, Rissanen 86, Quinlan&Rivest 89]. Since MDL uses a different tradeoff between accuracy and coverage than that used by Laplace accuracy, it is difficult to compare results using this function to those presented earlier. Instead, we modify Laplace accuracy to include a preference for short rules.

Laplace-depth is a modification of Laplace accuracy that includes a preference for short rules by setting k, the parameter that controls the bias on the prior distribution, to be proportional to the rule's length. If we let $L(r)$ denote rule length, then the formula for Laplace-depth, $\mathcal{LD}(r)$, is as follows:

$$\mathcal{LD}(r) = \frac{|E_+(r)| + L(r) \cdot \mu}{|E(r)| + L(r)}$$

.

By setting k to the rule length, Laplace-depth assumes a prior distribution with lower variances for longer rules. Since lower variances result in more conservative accuracy estimates, Laplace-depth assigns lower scores to longer rules.

Figure 4.5 compares the performance of Laplace-depth (solid lines) and Laplace accuracy (dotted lines) using beam search. Laplace-depth achieves the desired effect on the Promoters database. On this database, the accuracy of the rules learned increases essentially monotonically with increased search, and the rules learned with Laplace-depth consistently have higher accuracy. However, the learning curves for Laplace-depth only exhibit this ideal behavior on the Iris, Monk3, Mushroom, and Promoters databases. While the shape of the curve is improved in most databases, the curves often fail to be strictly monotonic. More importantly, the learning curves for Laplace-depth do not consistently result in more accurate rules. For a beam width of 512, the inductive performance of Laplace-depth is better on eight databases, worse

on eight databases, and the same on two.

The shape of the learning curve is very important for conducting larger searches. The curves in Figure 4.5 suggest that additional search will reduce the accuracy of the rules learned with Laplace accuracy while the performance of Laplace-depth will remain relatively stable. Figures 4.6 and 4.7 show that this prediction is correct. Figure 4.6 shows that exhaustive search using Laplace accuracy is outperformed by beam-512 search on most database. Figure 4.7 shows that exhaustive search and beam-512 search perform similarly when learning with Laplace-depth. Each algorithm outperforms the other on approximately half the databases.

Figure 4.8 compares the performance of Laplace-depth and Laplace accuracy for exhaustive search. While the results show each function outperforms the other in about one-half the databases, the performance improvements afforded by Laplace-depth are much greater. Laplace-depth reduces accuracy by more than two percentage points on only one database, while it increases accuracy by more than two percentage points on six databases. Laplace-depth is the better evaluation function for large-scale search.

Figure 4.9 shows that exhaustive search easily outperforms greedy search. The performance of greedy search is presented using Laplace accuracy because greedy search does not perform well with Laplace-depth. Massive search performs better on almost every database, frequently finding rules that are five percentage points higher than those found with greedy search.

Exhaustive search using Laplace-depth does not provide a clear performance gain over layered search with Laplace accuracy. Figure 4.10 shows that layered search performs better than exhaustive search with layered search performing better on ten databases and exhaustive search performing better in six databases.

While Laplace-depth reduces oversearching, it does not eliminate it. The problem may be that Laplace-depth makes the wrong tradeoff among rule complexity, data accuracy, and data coverage. It is also possible that there are additional factors

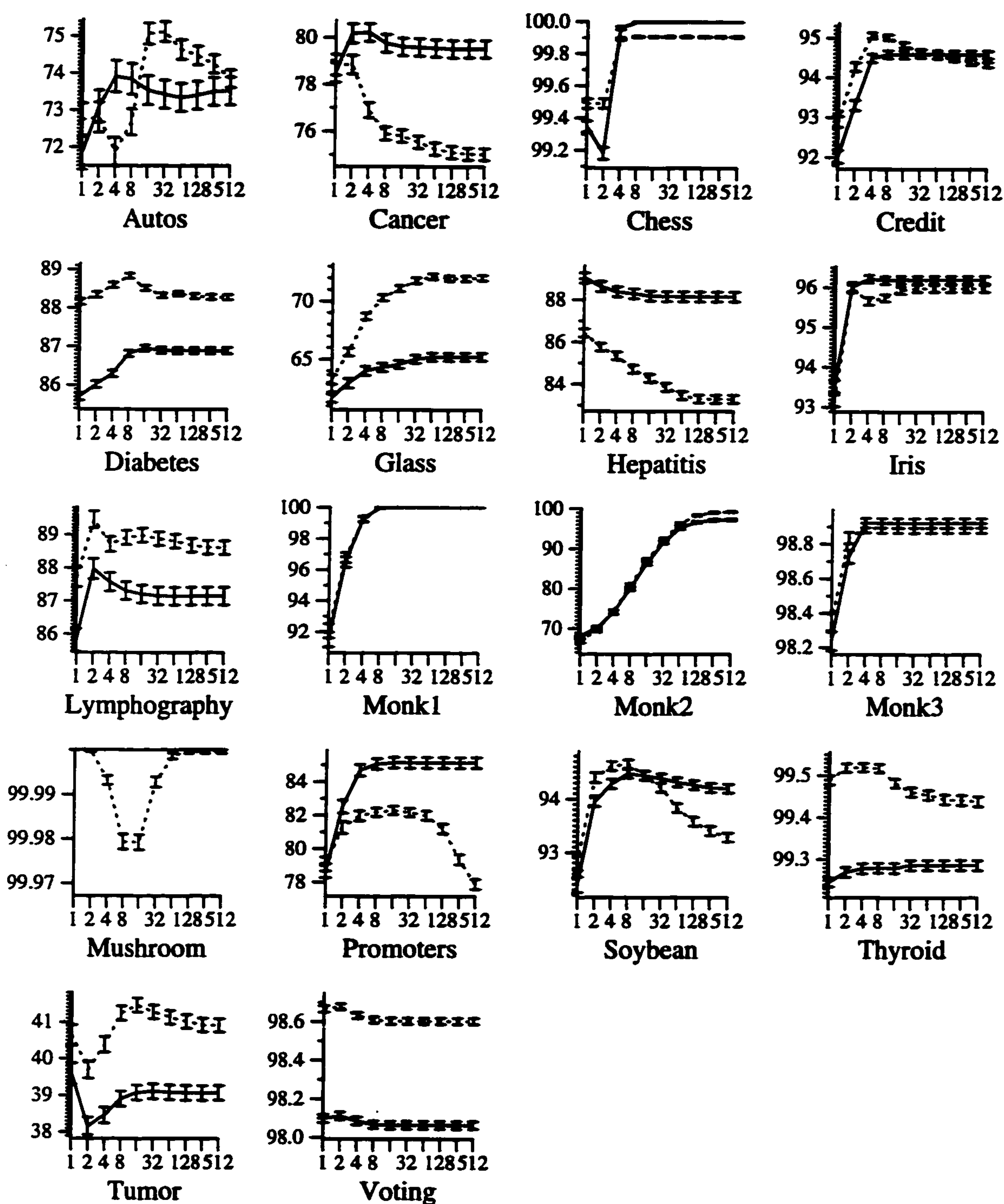


Figure 4.5: Comparison of the inductive performance Laplace-depth (solid lines) and Laplace accuracy (dotted lines). While the learning curves for Laplace-depth show less oversearching, they fail to outperform Laplace accuracy for beam-512 search.

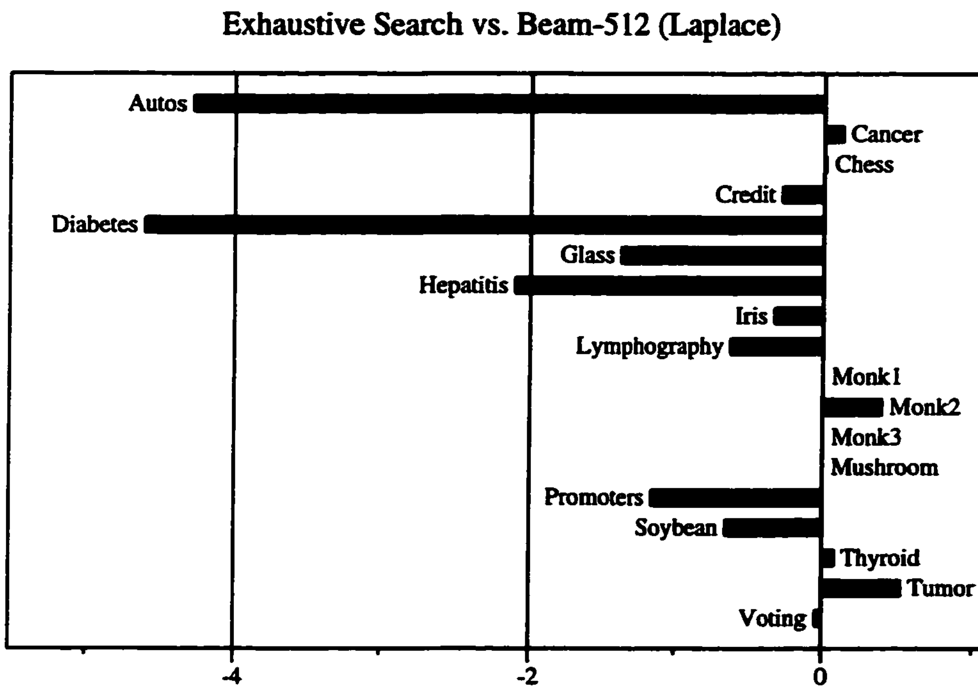


Figure 4.6: Difference in test accuracy between exhaustive search and beam-512 search when learning with Laplace accuracy. The graph shows that Laplace accuracy performs worse for exhaustive search.

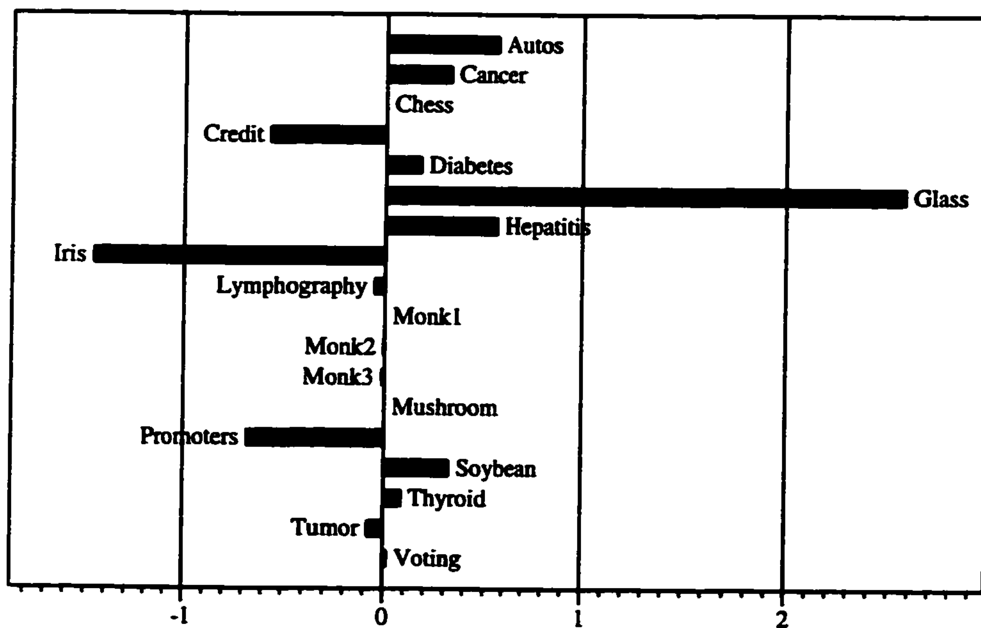


Figure 4.7: Difference in test accuracy between exhaustive search and beam-512 search when learning with Laplace-depth. The inductive performance of Laplace-depth remains relatively stable with additional search.

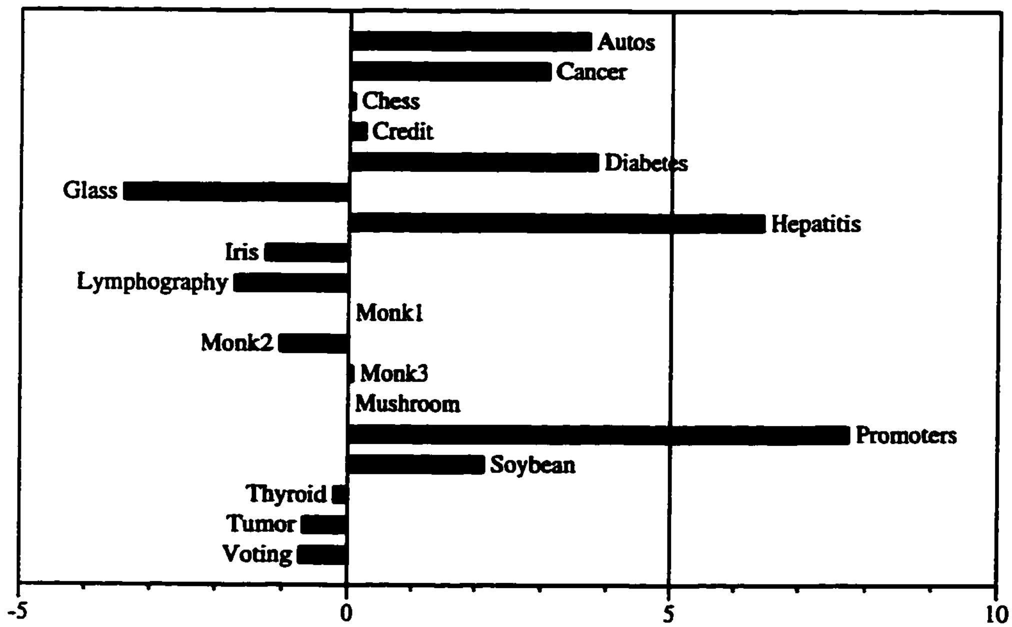


Figure 4.8: Comparison of Laplace-depth and Laplace accuracy for exhaustive search. While each function outperforms the other in roughly one-half the databases, the accuracy differences are much greater when Laplace-depth performs better.

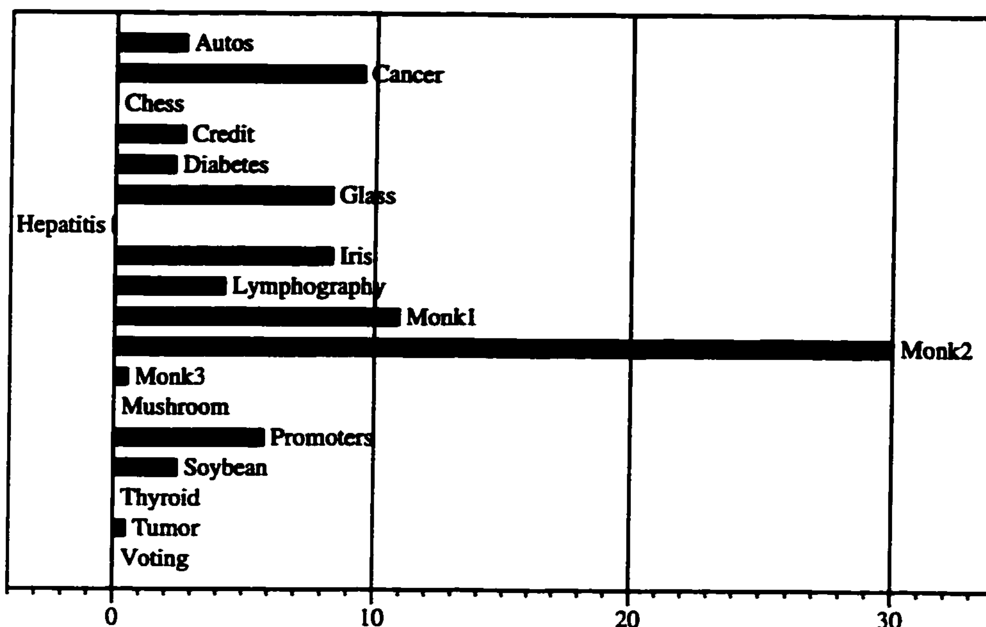


Figure 4.9: Comparison of exhaustive search using Laplace-depth and greedy search using Laplace accuracy. Exhaustive search performs better on almost every database.

that must be taken into account in order to properly evaluate a rule. However, the partial success of Laplace-depth suggests that further consideration of rule evaluation functions may allow massive search to achieve its full potential.

4.3 *Related Work*

Webb [1993] first identified the oversearching problem. Quinlan and Cameron-Jones [1995] conducted a detailed analysis of oversearching and concluded it is a direct result of extensive search. They advocate using a layered-search technique that limits the amount of search performed. Our results suggest that it may be possible to avoid oversearching without limiting search.

Quinlan and Cameron-Jones argued that oversearching is fundamentally different from overfitting by observing similar complexities for the *classifiers* learned by massive search and greedy search when combined with a simple covering algorithm.

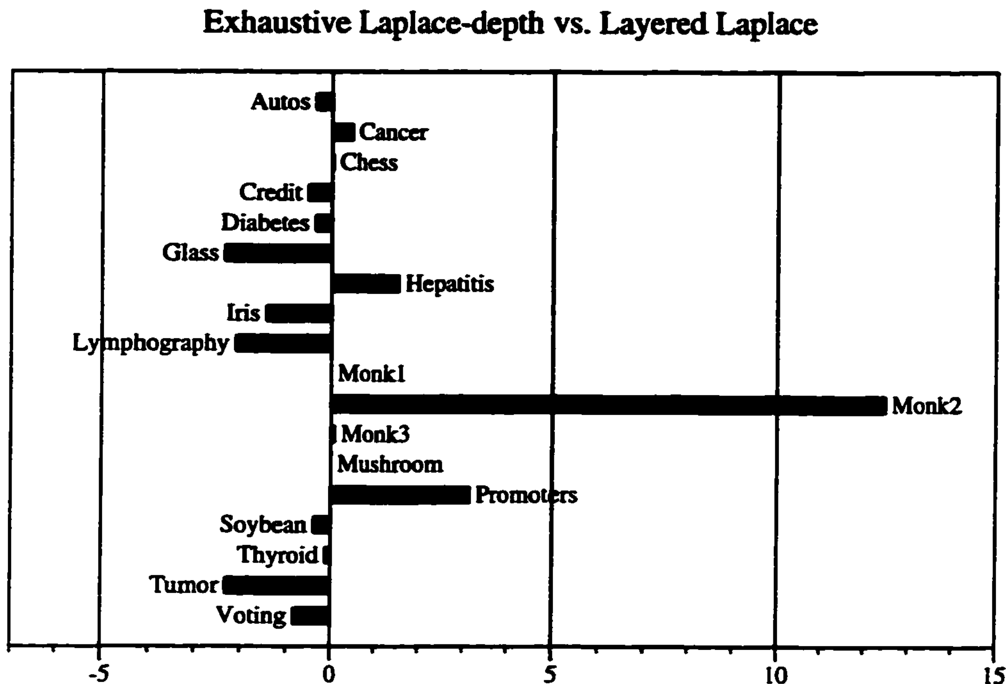


Figure 4.10: Comparison of exhaustive search and layered search using Laplace-depth. Layered search outperforms exhaustive search on most databases, but exhaustive search affords a large advantage on the Monk2 database.

While they note that the classifiers learned using massive search have longer rules, they argue that the additional rule complexity is counterbalanced by the massive-search classifiers containing fewer rules. However, this argument fails to recognize the importance of overfitting for *rule learning* and, as will be elaborated in Chapter 6, fails to recognize how the quality of the individual rules contribute to overall classifier performance.

Quinlan and Cameron-Jones' argument that oversearching is a natural consequence of visiting more rules is motivated by Blumer *et al.'s* [1987] treatment of Valiant's PAC-learning model [Valiant 84]. PAC-learning theory shows that the quality of the rules learned on a database decreases with the size of the hypothesis space. If we are willing to equate the number of rules visited to the size of the hypothesis

space, then PAC-learning theory would provide a theoretical justification for visiting fewer rules. However, equating the number of hypotheses and the number of rules visited is not appropriate. The hypothesis space used for estimating inductive performance based on PAC-learning theory should include any hypotheses that could *potentially* be returned by the algorithm. Since the set of rules that can potentially be returned by massive search and layered search is the same, their PAC analysis should be identical.

We can further illustrate the problems with a PAC-type analysis for oversearching by considering rule pruning. Imagine adding new pruning axioms to Brute that allow it to search one thousand times more efficiently. The new Brute would learn the same rules as the old Brute but will visit one thousand times fewer rules. However, if we equate the number of rules visited to the size of the hypothesis space, PAC-learning theory suggests that the new Brute would have better inductive performance even though it was finding the exact same rules.

Vapnik, a pioneer in PAC-learning theory, actually advocates a model very similar to Brute. Vapnik's structural risk-minimization (SRM) theory [Vapnik 95] asserts that the expected error of any classifier is the sum of its actual error and the error implied by the complexity of the smallest hypothesis space in which the rule fits. SRM suggests that classifier evaluation functions should take both classifier accuracy and classifier complexity into account. While this theory does not directly apply to rule learning, it does give indirect support for incorporating complexity into Laplace accuracy. Brute can be considered a practical implementation of Vapnik's SRM model.

4.4 Summary

Webb [1993] and Quinlan and Cameron-Jones [1995] have noted that inductive performance does not necessarily increase with additional search. Quinlan and Cameron-

Jones have offered an explanation of this phenomenon based on *fluke rules* that have high scores but poor inductive performance. While Quinlan and Cameron-Jones use this argument to advocate limited search, the argument also suggests oversearching can be eliminated by designing better evaluation functions.

This chapter takes a first step towards developing better evaluation functions for massive search. By analyzing the actual rules found by varying degrees of search, we found that oversearching can be partially explained by high-coverage overfitting. High-coverage overfitting is the learning of overly-complex rules that achieve high accuracy and high coverage by forming the conjunction of many high-coverage tests. High-coverage overfitting, like other types of overfitting, can be reduced by adding a preference for short rules. We extended Laplace accuracy to include a preference for short rules and evaluated the new function, Laplace-depth, experimentally. The new function reduces oversearching, particularly for exhaustive search, but fails to eliminate it entirely.

While massive search is shown to perform better than greedy search, Quinlan and Cameron-Jones' layered-search algorithm still outperforms massive search. The partial success of Laplace-depth suggests that further research into better evaluation functions may result in unlocking massive search's potential. However, even if Quinlan and Cameron-Jones are correct and oversearching is a direct consequence of large-scale search, the techniques presented in this thesis apply equally well to layered search.

Chapter 5

DATA MINING

Brute's performance in rule learning suggests it may be useful in other machine-learning tasks that require a rule-learning component. This chapter considers applying Brute to data mining. Data mining is the automatic extraction of useful information from large databases. This chapter presents the data-mining problem, discusses limitations of existing solutions, and shows how Brute offers a better solution. The next section motivates the data-mining problem using a real-world Boeing manufacturing application. The following section defines the data-mining problem. Section 5.3 describes existing solutions and discusses their limitations. Section 5.4 describes how Brute solves the data-mining problem and presents empirical results comparing Brute to existing methods. Section 5.5 discusses related work before concluding.

5.1 Boeing Manufacturing Domain

Boeing has a semi-automated work cell within one of its manufacturing facilities which automatically collects data about its day-to-day operations. This data contains information about various inefficiencies that may exist within the factory. The goal of data mining is to automatically extract this implicit information and make it available to factory engineers [Riddle *et al.* 92].

Each work cell consists of multiple automated workstations. The work cell routes nests of parts through task-specific workstations to formulate various hard goods.[1]

[1] We can only provide a limited amount of concrete detail regarding the learning task due to

When the cell encounters a problem, an alarm sounds. The work cell is staffed by a cell operator who addresses problems that arise. As alarm handling often involves processing delays, the cost of recovery increases product costs.

Information about the nests processed, the alarms triggered, and maintenance histories are stored in computerized operation records. These operation records implicitly contain vital information about how the factory can be improved. The goal of applying data-mining to this database is to extract useful patterns that suggest factory improvements. The types of patterns Boeing would like to extract include "alarm B sounds thirty minutes after alarm A," "alarm C is twice as likely to sound on material X," and alarm cascades. Alarm cascades are when one failure causes a chain reaction of related alarms. Boeing is also interested in predicting rejection tags, parts which do not pass quality inspection. For example, Boeing hopes to find patterns of the form "a rejection tag is four times as likely on parts made of batch C of material X." This information is useful for reducing the number of rejected parts; thus, saving money in wasted materials and manufacturing delays.

The Boeing data-mining application is an instance of the rule-learning problem because each of the patterns they wish to extract can be represented using if-then rules. For example, the pattern "a rejection tag is four times as likely on parts made of batch C of material X" can be represented as the if-then rule:

$$\textit{If material} = X \wedge \textit{batch} = C \textit{ then tag} = \textit{reject}, (\textit{4x as likely}).$$

This pattern differs from the rules learned by Brute by the addition of the phrase (*4x as likely*) to describe rule performance. Assuming Brute can find interesting rules, annotating the rule found with performance information is straightforward.

Data mining differs from rule learning because data-mining systems are expected to return multiple interesting patterns rather than the single rule returned by rule-

Boeing's non-disclosure requirements. While all experimental results reported were performed on actual Boeing data, all rules discussed within this chapter were created for pedagogical purposes and are not representative of the patterns found in the real Boeing databases.

learning algorithms. The focus of this chapter will be extending Brute to learn multiple rules. But first, we formally define the data-mining problem.

5.2 Problem Definition

The data-mining problem is the problem of inducing a set of interesting rules from a database. Data mining can be defined by extending the definition of rule learning to include learning multiple rules.

The natural definition of data mining requires extracting *all* interesting rules from a database. This definition requires a method for judging which rules are interesting. Interestingness can be judged using a utility function combined with a threshold that delineates interesting and uninteresting rules. However, choosing an appropriate threshold can be very difficult — too high a threshold and interesting rules are missed, too low a threshold and many uninteresting rules are returned.

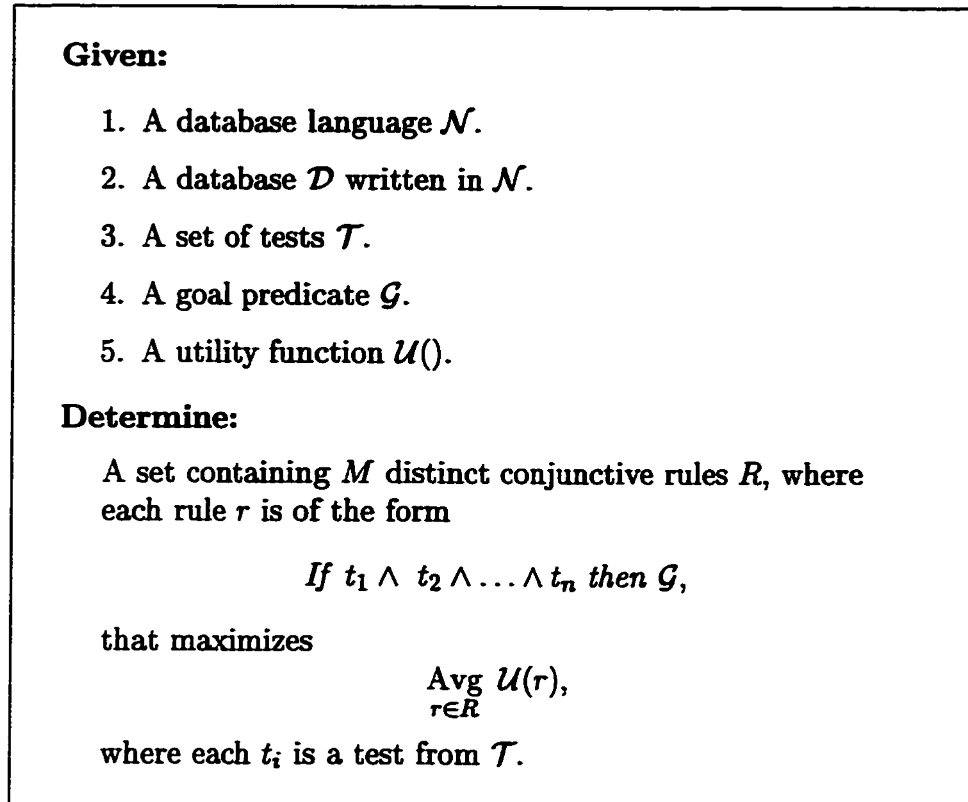
A more practical definition of data mining requires extracting the best M rules in order of increasing interestingness. The number of rules to return can be set based on the number of rules the user is willing to analyze. By returning the rules in order of interestingness, the user can determine the dividing line between interesting and uninteresting patterns based on the rules themselves.

Figure 5.1 presents our definition of the data-mining problem. The input to data mining is identical to the input to rule learning. The output of data mining is a set of M distinct rules that maximizes average rule utility. The returned rules are required to be distinct to ensure each rule conveys a different piece of information. For instance, the rules

$$\textit{If material} = X \wedge \textit{batch} = C \textit{ then tag} = \textit{reject}, (\textit{4x as likely})$$

and

$$\textit{If material} = X \wedge \textit{batch} = C \wedge \textit{day} = \textit{Monday then tag} = \textit{reject}, (\textit{4x as likely})$$



Given:

1. A database language $\mathcal{N}$.
2. A database $\mathcal{D}$ written in $\mathcal{N}$.
3. A set of tests $\mathcal{T}$.
4. A goal predicate $\mathcal{G}$.
5. A utility function $\mathcal{U}()$.

Determine:

A set containing M distinct conjunctive rules R, where each rule r is of the form

$$\textit{If } t_1 \wedge\ t_2 \wedge \ldots \wedge t_n \textit{ then } \mathcal{G},$$

that maximizes

$$\operatorname*{Avg}_{r \in R} \mathcal{U}(r),$$

where each t_i is a test from $\mathcal{T}$.

Figure 5.1: High-level description of the data-mining problem.

are not distinct because we typically assume that a pattern holds for all specializations of a rule unless given information to the contrary. Avoiding trivial rule variations is important when learning rule sets. If there are many trivial variations of the best rule, the best M rules may be restatements of the single best rule.

It is difficult to precisely delineate between similar and distinct rules. Rather than provide an exact definition, the following sections discuss different types of rule similarities and how they can be avoided. The set of similarities we consider is not exhaustive but is representative of the similarities encountered on real data. We leave a formal definition of rule similarity for future work.

The utility function for data mining varies from application to application. For the Boeing domain, the utility of a rule is determined by its potential impact on

the factory. While a rule's impact depends on both its accuracy and coverage, this dependency is difficult to model and is beyond the scope of this thesis. Instead, we evaluate performance on the Boeing database using both accuracy and rule coverage. We will consider an algorithm better than another if it simultaneously learns both more accurate and higher-coverage rules.

5.3 Existing Solutions

5.3.1 Decision Trees

Data mining is often performed using decision-tree algorithms such as CART [Breiman *et al.* 84] and C4.5 [Quinlan 93]. Decision-tree algorithms were not designed for data mining but for learning classifiers. Classifiers are functions that predict the goal attribute's value from the values of the remaining attributes.

A decision tree is a tree-structured classifier. The nodes of a decision tree contain tests from the same test set used by Brute. The tests are used to filter each example down to one of the leaves which contain the predicted value for the goal attribute. Figure 5.2 shows a sample decision tree for predicting when parts fail inspection.

Decision-tree algorithms learn decision trees using a recursive tree-growing process. Each test is evaluated on the training data using a *test-selection function*. The test-selection function assigns each test a score based on how well it partitions the data. The test with highest score is selected and placed at the root of the tree. The subtrees of each leaf are then grown recursively by applying the same algorithm to the examples in each leaf. The algorithm terminates when the current node contains either all positive or all negative examples.

Decision-tree algorithms are used for data mining because they have good inductive performance and because decision trees can be easily converted into rules [Quinlan 87]. Each leaf of a decision tree is equivalent to a rule whose antecedent contains all the tests on the path from the root to the leaf and whose consequent predicts the

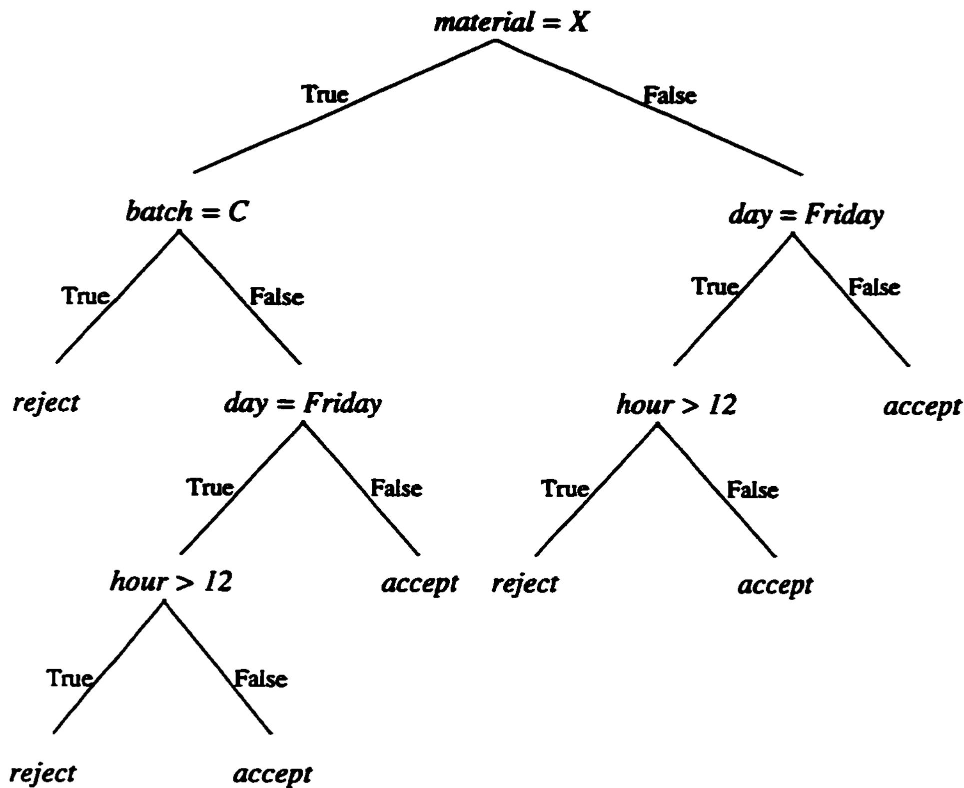


Figure 5.2: Hypothetical decision tree for predicting when parts manufactured in a Boeing factory fail inspection.

leaf's value for the goal attribute. Figure 5.3 shows the rules implicit in the decision tree of Figure 5.2 that predict *reject*, the desired value of the goal attribute. Decision-tree algorithms can be used for data mining by first building a decision tree and then extracting the relevant rules.

The decision-tree approach to data mining has several problems because decision-tree algorithms were not originally designed for data mining. The foremost problem is that the test-selection functions used for tree building can prevent finding the best rules. Classical test-selection functions, such as information gain [Quinlan 86] or the Gini Index [Breiman *et al.* 84], have the following form:

$$\max_{t \in \mathcal{T}} \left(\mathcal{C}_{\mathcal{D}}(t) \cdot P(\mathcal{A}_{\mathcal{D}}(t)) + \mathcal{C}_{\mathcal{D}}(\neg t) \cdot P(\mathcal{A}_{\mathcal{D}}(\neg t)) \right),$$

If material $= X \wedge$ *batch* $= C$ *then tag* $=$ *reject*, (*4x as likely*),
If material $\neq X \wedge$ *day* $=$ *Friday* $\wedge$ *hour* > 12 *then tag* $=$ *reject*, (*3x as likely*),
If batch $\neq C \wedge$ *day* $=$ *Friday* $\wedge$ *hour* > 12 *then tag* $=$ *reject*, (*3x as likely*).

Figure 5.3: Rules extracted from the decision tree of Figure 5.2. A rule is created for each path in the decision tree that leads to the desired value of the goal attribute, *reject*.

where $P()$ measures the purity of a node, and $\mathcal{A}_{\mathcal{D}}(t)$ and $\mathcal{C}_{\mathcal{D}}(t)$ denote the data accuracy and data coverage of a test on the examples matching the current node. This function evaluates each test by averaging the purity of each branch, weighted by the number of examples filtered to each branch. Although the exact measure of purity varies from one algorithm to another, standard measures exhibit similar behavior in practice [Breiman *et al.* 84, Mingers 89].

Test-selection functions that compute a weighted average across branches can cause decision-tree algorithms to overlook the best rules. These functions prefer splits that achieve good performance in each branch over splits that generate better performance on a single branch. Yet, a single high-purity branch may convey a more interesting pattern. Figure 5.4 illustrates this bias. In the figure, `P` and `N` denote the number of positive and negative examples that satisfy each test. Adding the test T_1 is preferable because it yields a branch with 100% accuracy on the training data. Standard selection functions weight the purity of each branch of the test. Since the branch where T_1 is false has very low purity, the overall score for test T_1 is low. Standard selection functions incorrectly prefer test T_2 because it has greater than average purity for both of its branches.

A second problem with decision trees is that they only provide a single explanation for each training example even when multiple explanations are appropriate. The rules learned by decision-tree algorithms are disjoint; no example can match the antecedent of more than one rule. As a result, decision-tree algorithms cannot learn overlapping

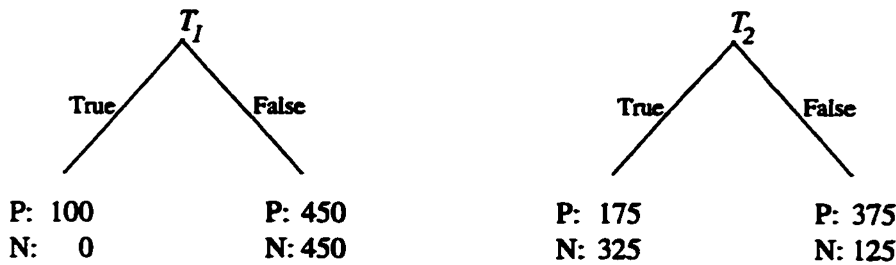


Figure 5.4: An example illustrating how standard test-selection functions can overlook good rules. Both information gain and the Gini Index prefer test T_2 over test T_1, whereas the left side of T_1 is the most interesting rule.

rule combinations such as

$$\begin{aligned}&\textit{If material} = X \wedge \textit{batch} = C \textit{ then tag} = \textit{reject}, (\textit{4x as likely}),\\ &\textit{If day} = \textit{Friday} \wedge \textit{hour} > 12 \textit{ then tag} = \textit{reject}, (\textit{3x as likely}).\end{aligned}$$

These rules are overlapping because they can be true at the same time, parts of material "X" and batch "C" can be manufactured on Friday afternoons. Since decision-tree algorithms cannot learn overlapping rules, the best they can do is learn the following equivalent set of non-overlapping rules:

$$\begin{aligned}&\textit{If material} = X \wedge \textit{batch} = C \textit{ then tag} = \textit{reject}, (\textit{4x as likely}),\\ &\textit{If material} \neq X \wedge \textit{day} = \textit{Friday} \wedge \textit{hour} > 12 \textit{ then tag} = \textit{reject}, (\textit{3x as likely}),\\ &\textit{If batch} \neq C \wedge \textit{day} = \textit{Friday} \wedge \textit{hour} > 12 \textit{ then tag} = \textit{reject}, (\textit{3x as likely}).\end{aligned}$$

Figure 5.2 showed the decision tree that generates these three rules. While it is possible to remove the extraneous conjuncts using postprocessing, the extra conjuncts make the rules more difficult to find in the initial tree-growing step. The problem is the more conjuncts prepended in front of a rule, the fewer examples left to evaluate the remaining conjuncts. The fewer examples matched by a rule, the more difficult it is to learn using decision trees. As a result, decision trees have difficulty learning overlapping rules.

A third problem with decision trees arises from their use of greedy search. In addition to reducing inductive performance, greedy search creates uncertainty as to

whether there is more information in the database that may be useful. The relative cost of applying machine-learning methods is tiny compared to the potential benefits of successful data mining. Learning a single extra rule can be worth millions of dollars. When the dozen or so rules learned by a decision-tree algorithm have been analyzed by factory personnel, there remains the question as to whether there are additional useful rules implicit in the database that were missed by greedy search. Lack of confidence in heuristic search has led researchers to try various ad hoc methods for learning additional rules including resampling and removing attributes for which rules have already been learned.

5.3.2 Gold-digger

Decision trees are not appropriate for data mining because they were originally designed for classification. The difficulties decision trees have with data mining therefore may not be related to using greedy search. This section proposes a greedy algorithm called Gold-digger [Riddle *et al.* 94], which was specifically designed for data mining. Although developed independently, Gold-digger shares structure with a variety of decision-list learning algorithms including AQ [Michalski 69], Greedy3 [Pagallo&Haussler 90], and CN2 [Clark&Boswell 91]. Gold-digger combines a greedy rule-learning algorithm with a covering algorithm for learning multiple rules. Gold-digger's rule-learning algorithm is the greedy rule-learning algorithm presented in Chapter 2 extended to reduce overfitting. Gold-digger reduces overfitting in a different manner than Brute. Instead of learning with Laplace-depth, Gold-digger learns with Laplace accuracy and adds a postpruning phase that limits rule complexity.

Gold-digger's postpruning phase is very similar to the tree-pruning technique used by CART [Breiman *et al.* 84]. The training data is randomly partitioned, 70% for learning and 30% for pruning. Rule learning proceeds as normal on the learning data. Postpruning re-evaluates the learned rule and searches for the subset of the rule's antecedent which is likely to have the best inductive performance. The pruning

data is used to give an independent assessment of each subset's predictive ability.

Gold-digger's postpruning algorithm uses greedy search to find the best subset. The postpruning algorithm evaluates the rule learned and the rules that can be formed by removing exactly one conjunct. If removing any conjunct improves pruning-data accuracy, the conjunct is removed and the process is repeated. Postpruning continues until removing additional conjuncts does not improve pruning-data performance. By removing conjuncts that do not improve accuracy on an independent test set, postpruning reduces overfitting.

Gold-digger learns multiple rules by incorporating its rule-learning algorithm into a greedy covering algorithm. Gold-digger's covering algorithm begins by finding the single-best rule according to its greedy rule-learning algorithm. The covering algorithm then removes from the training data any positive examples matching the best rule and re-invokes the rule-learning algorithm on the remaining examples. Removing the positive examples covered by the first rule guarantees that the second greedy search will learn a different rule. This process is repeated to learn additional rules until either no good rules are found or all positive training examples have been covered. Figure 5.5 presents the complete algorithm for Gold-digger.

Gold-digger should be more effective than decision trees for data mining because it uses an evaluation function specifically designed for data mining. The improved evaluation function prevents Gold-digger from missing good rules in the path of its greedy search. We tested this hypothesis by comparing the data mining performance of Gold-digger and C4.5 on two Boeing manufacturing databases. Both databases contain operation records from a Boeing factory, but the organization of each database is tailored to a particular learning task. The first database, the Occupancy-time database, contains information about how much time parts in the factory spend at each machine. This database is used to learn when parts take an unusually long time to move from machine to machine and is useful for increasing factory throughput. The second database, the Failed-parts database, contains information about parts

```
FUNCTION PostpruneRule(NewRule, PruneData):RULE
    BestRule = NewRule
    BestScore = ComputeScore(NewRule, PruneData)
    REPEAT
        CurrentRule = BestRule
        FOREACH Conjunct IN Conjuncts(CurrentRule) DO
            PrunedRule = RemoveConjunct(CurrentRule, Conjunct)
            PrunedScore = ComputeScore(PrunedRule, PruneData)
            IF (PrunedScore > BestScore) THEN
                BestRule = PrunedRule
                BestScore = PrunedScore
            ENDIF
        END
    UNTIL BestRule = CurrentRule OR BestRule = EmptyRule(Goal)
    RETURN BestRule
END

FUNCTION GoldDigger(Goal, Tests, Database):RULESET
    TrainData = RandomlySelect(Database, 70%)
    PruneData = Database - TrainData
    Ruleset = ∅
    REPEAT
        Rule = GreedyLearn(Goal, Tests, TrainData)
        PrunedRule = PostpruneRule(Rule, PruneData)
        Ruleset = Ruleset ∪ PrunedRule
        TrainData = TrainData - MatchedPositives(PrunedRule, TrainData)
    UNTIL Positives(TrainData) = ∅ OR PrunedRule = EmptyRule(Goal)
    RETURN Ruleset
END
```

Figure 5.5: Pseudocode for Gold-digger, a greedy algorithm specifically designed for data mining. Gold-digger use the GreedyLearn procedure presented in Figure 2.4.

being manufactured in the factory and whether they pass final inspection. The goal of mining this database is to learn when parts fail inspection to increase factory yield and reduce costs.

Data-mining performance is measured by the average utility of the rules learned. Since utility in the Boeing domain increases monotonically with both rule accuracy and rule coverage, we evaluate each algorithm using both the average accuracy and the average coverage of the rules returned. Table 5.1 shows the results of running Gold-digger and C4.5 to find the best ten rules on the two Boeing databases. While the goal is to learn ten rules, Gold-digger and C4.5 often fail to learn that many. When this occurs, we evaluate each algorithm only on the rules learned. This can create an unfair situation since it is easier to learn three good rules than ten good rules. As a result, Table 5.1 presents two different numbers for Gold-digger's performance: Gold-digger's performance when learning up to ten rules, and Gold-digger's performance when restricted to learning the same number of rules as C4.5.

On the Occupancy-time database both algorithms found rules with similar accuracy and coverage. Gold-digger found slightly more accurate rules, and C4.5 found slightly higher coverage rules. However, Gold-digger was able to extract ten interesting patterns while C4.5 could only find about six. On the Failed-parts database Gold-digger finds rules with both substantially higher accuracy and higher coverage than those found by C4.5. Again, Gold-digger finds many more interesting patterns.

While Gold-digger performs well, it inherits many of decision-tree's problems. Like decision trees, Gold-digger's covering algorithm has difficulty learning overlapping rules. While we considered a variety of ad hoc methods for learning overlapping rules within Gold-digger, none of them could provide the assurances offered by massive search. Once we realized that massive search was plausible, we dedicated our efforts to that approach.

Table 5.1: Comparison of Gold-digger and C4.5 for mining two Boeing manufacturing databases. The table shows the average accuracy and average coverage of the rules returned by each algorithm. Since Gold-digger and C4.5 do not always return ten rules, averages are shown for Gold-digger learning the same number of rules as learned by C4.5. While Gold-digger finds many interesting rules on both databases, C4.5 only finds an average of 6.1 useful rules on the Occupancy-time database and barely finds any rules on the Failed-parts database.

Occupancy-Time Database

(Base rate = 11.1%)

Algorithm	Average Test Accuracy	Average Test Coverage	Rules Returned
Gold-digger – *10 rules*	51.1	5.3	10.0
C4.5 – *10 rules*	51.8	9.1	6.1
Gold-digger – *# rules C4.5*	54.8	6.1	6.1

Failed-Parts Database

(Base rate = 7.2%)

Algorithm	Average Test Accuracy	Average Test Coverage	Rules Returned
Gold-digger – *10 rules*	12.4	2.3	7.6
C4.5†	1.2	0.5	1.4
Gold-digger – *# rules C4.5*	14.5	2.3	1.4

† Failed to learn any rules on 40% of the random trials.

5.4 Data Mining with Brute

Brute can be extended for data mining by expanding the number of rules it returns. Rather than remembering the single best rule, the modified algorithm remembers the best M rules. While this extension is straightforward, there are two subtleties that must be considered. First, simply returning the M best rules is likely to result in many similar rules being returned. Brute needs additional mechanisms to ensure the rules it returns are distinct. Second, the pruning rules described in Chapter 3 do not necessarily carry over to learning multiple rules. The next section discusses how Brute prevents redundancy when learning multiple rules, and the following section discusses the implications for rule pruning.

5.4.1 Preventing Redundancy

The definition of data mining requires the returned rules convey different information to the end user. This section considers three different types of redundancy and describes extensions to Brute for handling each of them.

Trivial Specializations

Each rule describes a pattern that occurs whenever the rule's antecedent holds. We normally assume this pattern holds for ***all*** examples matching the rule. When given the rule,

$$\textit{If day} = \textit{Friday} \wedge \textit{hour} > 12 \textit{ then tag} = \textit{reject}, (\textit{3x as likely}), \tag{5.1}$$

we typically assume that ***any*** Friday afternoon, parts are three times more likely to fail inspection. As a result of this assumption, the rule

$$\textit{If day} = \textit{Friday} \wedge \textit{hour} > 12 \wedge \textit{month} = \textit{May then tag} = \textit{reject}, (\textit{3x as likely}) \tag{5.2}$$

is redundant because it conveys information already implied by the first rule. On the other hand, the rule

$$\textit{If } day = Friday \wedge hour > 12 \wedge summer = yes \textit{ then } tag = reject, (5x \textit{ as likely}) \quad (5.3)$$

is interesting because it indicates the failure rate is even higher for Friday afternoons in the summer. Both rule 5.2 and rule 5.3 are specializations of rule 5.1. Rule 5.2 is uninteresting because the predicted failure rate is the same as without the extra conjunct, *month = May*, while rule 5.3 is interesting because the predicted failure rate differs from that of rule 5.1.

We define a *trivial specialization* of a rule to be any specialization with similar predicted accuracy. Trivial specializations, if left unchecked, can seriously degrade the information returned by Brute. If Brute were asked to return the ten best rules for the example above, it is likely to return rule 5.1 along with nine specializations of rule 5.1 with each of the non-summer months added. While Brute returns ten rules, they all describe the same pattern.

Detecting trivial specializations is complicated by statistical variations and accuracy estimation. Even though rule 5.2 is a trivial specialization of rule 5.1, it is unlikely both rules will have exactly the same data accuracy because of sampling variances. Furthermore, rule 5.2 has greater complexity and will cover fewer examples than rule 5.1. These differences imply that rule 5.2, although expressing the same pattern, will be given a lower accuracy estimate. While the accuracy estimate is reduced, it is often not reduced enough to prevent trivial specializations from appearing in the final rule set. Trivial specializations can be detected using a χ^2 test to reject the hypothesis that the accuracy of a specialization is the same as the rule it specializes. Brute checks for trivial specializations using a χ^2 test at a confidence level of $p = 0.1$.

Brute incorporates its test for trivial specializations into its search algorithm to ensure that it learns the best M distinct rules. One approach would be to apply the

test whenever a new rule is added to the rule set. In this approach, each new rule that is among the best M seen thus far is compared against each of the other $M-1$ current best rules. If the new rule is a trivial specialization of an existing rule, it is discarded. If any trivial specialization of the new rule already exists in the current best rules, they are also discarded. In this way, Brute could ensure the current M rules never contain two rules that are trivial specializations of each other.

Unfortunately, this approach makes branch-and-bound pruning difficult. When learning multiple rules, branch-and-bound pruning removes any subtree whose maximum score is less than the current M^{th} best rule. The correctness of branch-and-bound pruning depends on the score of the Mth best rule strictly increasing during the search. However, the above algorithm for checking specializations can cause the score of the Mth best rule to decrease by replacing the Mth best rule with a lower scoring generalization.

A better approach is to avoid adding trivial specializations to the current rule set. Brute checks if a rule is a trivial variation of any of its parents before adding it to the current rule set. Figure 5.6 shows Brute's algorithm for eliminating trivial specializations. The algorithm first checks to see if the rule being evaluated is one of the current best. If not, the expensive similarity check can be avoided. Otherwise, Brute computes the accuracy of each generalization of the rule that can be generated by removing a single conjunct. These accuracies are compared against the accuracy of the rule using a χ^2 test. If the rule is found to be a trivial specialization of one of its parents, it is not added to the current rule set.

Brute's algorithm for handling trivial specializations only prevents rules from being added to the current rule set and does not prune portions of the search space. Some learning problems require adding more than one conjunct to a rule before any improvement in accuracy can be detected. Since Brute does not prune the search space below trivial specializations, it can still uncover rules that require multiple conjuncts to be added before a difference in accuracy is observed. However, there

```
FUNCTION TrivialSpecialization(Rule, Database):BOOLEAN
    FOREACH Conjunct IN Conjuncts(Rule) DO
        ParentRule = RemoveConjunct(Rule, Conjunct)
        Accuracy = ComputeAccuracy(ParentRule, Database)
        ChiSquare = ComputeChiSquare(Accuracy, Rule)
        IF PValue(ChiSquare) < 0.1 THEN
            RETURN True
        ENDIF
    END
    RETURN False
END

PROCEDURE StoreRule(Rule, Ruleset, M, Database)
    IF Length(Ruleset) < M OR Rule.Score > Ruleset[M].Score THEN
        IF NOT TrivialSpecialization(Rule, Database) THEN
            OrderedInsert(Rule, Ruleset)
            Truncate(Ruleset, M)
        ENDIF
    ENDIF
END
```

Figure 5.6: Brute's algorithm for eliminating trivial specializations. Each rule is checked to see if it is a trivial specialization of one of its parents. If it is, the rule is not added to the list of current rules.

are special cases in which Brute can prove that all rules in a subtree will fail the trivial-specialization filter. These cases are discussed in Section 5.4.2.

Correlated Rules

Some of the attributes within a database may be correlated. Attribute correlations can result in learning multiple variants of a good rule. The following rules appear distinct:

If $day = Friday \wedge hour > 12 \wedge summer = yes$ *then* $tag = reject$, ($5x$ *as likely*),
If $day = Friday \wedge hour > 12 \wedge month = July$ *then* $tag = reject$, ($5x$ *as likely*),
If $day = Friday \wedge hour > 12 \wedge month = August$ *then* $tag = reject$, ($5x$ *as likely*).

However, the last two are trivial specializations of the first because, when combined with the facts "*month* $=$ *July* $\Rightarrow$ *summer* $=$ *Yes*" and "*month* $=$ *August* $\Rightarrow$ *summer* $=$ *Yes*," the first rule implies the second two.

The question of what rules to return when a correlation exists can be complicated. If John Smith is always the on-site manager Friday afternoons and John Smith only works Friday afternoons, then the following three rules are correlated:

If day $=$ *Friday* $\wedge$ *hour* $>$ *12 then tag* $=$ *reject*, (*4x as likely*),
If manager $=$ *Smith then tag* $=$ *reject*, (*4x as likely*),
If day $=$ *Friday* $\wedge$ *hour* $>$ *12* $\wedge$ *manager* $=$ *Smith then tag* $=$ *reject*, (*4x as likely*).

It is impossible to determine from the data which of these three rules is best. The difficulties that occur on Friday afternoon may be because the employees lose focus just before the weekend, because John Smith is a poor manager, or because John Smith loses focus just before the weekend. Since it is impossible to choose which rule is best, a data-mining system should return all three rules and information about the correlation. However, the rules should not be counted as three distinct rules but should count as a single pattern. When returning patterns that represent multiple rules, there is a question as to how the pattern should be ranked for comparing it with other rules. We assign each pattern the score of the highest-scoring rule in the pattern's rule set.

One method for avoiding rule correlations is to have the user provide a theory specifying known correlations. This approach requires the user to provide knowledge that may not be easily available and eliminates the possibility of learning previously unknown correlations.

The ideal approach is to learn correlations directly from the training data. However, learning correlations is a difficult problem in its own right [Agrawal *et al.* 93] and beyond the scope of this thesis. Instead, Brute uses a heuristic approach that significantly reduces the number of correlated rules.

Brute's approach is to identify correlated rules and to return only the best rule from each correlated set. Brute uses its subsumption pruning algorithm as a heuristic for identifying correlated rules. Subsumption pruning eliminates a rule's siblings which cover a subset of its positive examples and a superset of its negative examples. Rule pairs that meet the criteria for subsumption pruning are usually correlated. Let R_1 denote an arbitrary rule and R_2 denote a subsumed sibling of R_1. There are three cases of interest: R_2 covers fewer positive examples and mostly the same negative examples, R_2 covers more negative examples and mostly the same positive examples, and R_2 covers more positive examples and fewer negative examples. We consider each in turn.

When R_2 covers a subset of R_1's positives and mostly the same negative examples, then R_2 covers essentially a subset of the examples covered by R_1. The examples covered by R_2 may not be a strict subset since it may cover a few negative examples not covered by R_1. Since R_2 covers approximately a subset of the examples covered by R_1, these rules are correlated by $R_2 \Rightarrow R_1$. Subsumption pruning, by pruning the search space below R_2, prevents both of these correlated rules from appearing in the final rule set.

The inverse situation occurs when R_2 covers a superset of R_1's negatives and mostly the same positives. In this case, R_1 covers approximately a subset of the examples covered by R_2. These rules are therefore correlated by $R_1 \Rightarrow R_2$. Again, pruning R_2 is the correct behavior since R_1 has the higher score.

The final case is where R_2 covers both less positive examples and more negative examples. In this case we have neither $R_1 \Rightarrow R_2$ or $R_2 \Rightarrow R_1$. Therefore, it is not correct to prune R_2 because no correlation between the two rules exist. Unfortunately, subsumption pruning does prune R_2 in these cases. While this is not the ideal behavior, it is often not problematic because the large differences in both positive and negative coverage will result in R_2 having substantially lower predictive accuracy. The rules incorrectly pruned by this heuristic are typically low scoring and usually

do not make the M best rules for moderate values of M. The benefits of subsumption pruning outweigh this potential danger.

Subsumption pruning has two additional limitations. The first limitation is caused by the incompleteness of Brute's subsumption pruning. Since subsumption pruning only checks for subsumption among sibling rules, it can fail to find correlated rules that appear at distant portions of the search space. The second limitation results from subsumption pruning not being designed for eliminating correlated rules. Subsumption pruning cannot detect correlated rules for which the correlated rule matches a subset of *both* the positive and negative examples.

While subsumption pruning fails to detect certain classes of correlated rules, its ability to eliminate many correlations makes it worthwhile. Furthermore subsumption pruning significantly reduces the cost of massive search. While the downside of subsumption pruning is that it may prune uncorrelated rules that differ by a large number of positive and negative examples, these rules have low scores and are unlikely to be part of the final rule set.

While Brute's subsumption pruning helps prevent returning correlated rules, it throws away valuable information about rule correlations. Brute regenerates the rule correlations for each returned rule using the postprocessing algorithm shown in Figure 5.7. For each rule in the final rule set, the postprocessing algorithm considers all possible single test replacements for its existing conjuncts. Any single test replacement which generates a correlated rule is returned.

Similar Numeric Thresholds

The final type of rule similarity Brute handles occurs because of numerical attributes. Tests on numerical attributes compare the attribute's value with a numeric threshold. Rule performance is often not sensitive to minor changes in a rule's numeric thresholds. As a result, these minor changes produce rules with similar scores that convey little if any additional information. If left unchecked, these variations reduce

```
PROCEDURE GenerateCorrelatedRules(Ruleset, Tests, Database)
    FOREACH Rule IN Ruleset DO
        Rule.CorrelatedRules = ∅
        FOREACH Conjunct IN Conjuncts(Rule) DO
            FOREACH Test IN Tests DO
                NewRule = RemoveConjunct(Rule, Conjunct)
                NewRule = AddConjunct(NewRule, Test)
                IF Correlated(Rule, NewRule, Database) THEN
                    Rule.CorrelatedRules = Rule.CorrelatedRules ∪ NewRule
                ENDIF
            END
        END
    END
END
```

Figure 5.7: Postprocessing algorithm for generating sets of correlated rules from the individual rules returned by Brute's massive search algorithm. The algorithm finds any correlated rule that can be formed by making any single-test substitution.

the number of distinct patterns returned by Brute.

The difficulties with numerical attributes are illustrated by the following sequence of similar rules:

$$\begin{aligned}
&\textit{If } day = \textit{Friday} \wedge hour > 10 \textit{ then } tag = \textit{reject}, (4.3x \textit{ as likely}),\\
&\textit{If } day = \textit{Friday} \wedge hour > 11 \textit{ then } tag = \textit{reject}, (4.7x \textit{ as likely}),\\
&\textit{If } day = \textit{Friday} \wedge hour > 12 \textit{ then } tag = \textit{reject}, (5x \textit{ as likely}),\\
&\textit{If } day = \textit{Friday} \wedge hour > 13 \textit{ then } tag = \textit{reject}, (5x \textit{ as likely}),\\
&\textit{If } day = \textit{Friday} \wedge hour > 14 \textit{ then } tag = \textit{reject}, (5x \textit{ as likely}).
\end{aligned}$$

Each rule uses a slightly-different threshold for the *hour* attribute; therefore, each rule has slightly different predictive power. These rules are not distinct because their high scores are the result of the same phenomenon, an increased failure rate on Friday afternoons.

Brute avoids learning rules with similar thresholds by filtering out all but the best rule from each set with the same *test signature*. The test signature of a rule is the

rule's antecedent with each numeric threshold replaced with the wild-card character "?". The test signature for each rule in the above example is

$$day = Friday \wedge hour > ?.$$

Since these rules share the same test signature, Brute returns the rule from this set with the highest score. The rule with the highest score is the rule whose second conjunct is $hour > 12$ because it has the highest coverage of the three maximally-accurate rules.

Brute's similar-threshold filter is implemented as part of the rule-storage routine presented earlier. Figure 5.8 shows Brute's rule-storage routine with the new code for handling similar thresholds added in italics. The similar-threshold filter is only applied to rules which have already passed the trivial-specializations filter. The filter scans the existing rule set to determine if the new rule has the same test signature as an existing rule. If it does not, rule insertion proceeds as normal. Otherwise, the scores of the new rule and the rule with the same signature are compared. If the existing rule has a higher score, the new rule is filtered out. If the new rule has a higher score, it is added to the current rule set after removing the existing rule with the same signature.

5.4.2 Rule Pruning

Brute's existing pruning mechanisms work equally well for data mining. Branch-and-bound pruning must be slightly modified to prune subtrees whose upper-bound is less than the score of the Mth best rule. Subsumption pruning is desirable for eliminating redundancy and can be used without change. Depth-bound pruning is still appropriate for limiting search when a complete search is not feasible. While data mining does not require abandoning Brute's existing pruning mechanisms, it does provide an opportunity for additional pruning techniques. The remainder of

```
PROCEDURE StoreRule(Rule, Ruleset, M, Database)
    IF Length(Ruleset) < M OR Rule.Score > Ruleset[M].Score THEN
        IF NOT TrivialSpecialization(Rule, Database) THEN
            FOREACH StoredRule IN Ruleset DO
                IF TestSignature(Rule) = TestSignature(StoredRule) THEN
                    IF StoredRule.Score > Rule.Score THEN RETURN
                    Ruleset = Ruleset - StoredRule
                ENDIF
            END
            OrderedInsert(Rule, Ruleset)
            Truncate(Ruleset, M)
        ENDIF
    ENDIF
END
```

Figure 5.8: The function StoreRule() extended to filter rules with similar numeric thresholds. The new code is shown in italics. The filter ensures the final rule set contains only the best rule for each test signature.

this section describes a new pruning technique that prunes portions of the search space guaranteed only to contain trivial specializations.

Brute's trivial-specialization filter cannot be used directly to prune the search tree because adding seemingly irrelevant tests is necessary for learning some concepts, such as parity and exclusive-or. However, there are cases where adding an irrelevant test may cause all specializations of a rule to be trivial. An extreme example occurs when adding a test that is perfectly correlated with one of a rule's existing tests. This rule and all specializations of it will fail the trivial-specialization filter because removing one of the correlated tests will always produce a rule with identical accuracy and coverage.

Trivial-specialization pruning is based on the following theorem that asserts all specializations of a rule will be trivial if the rule does not differ from one of its parents by more than X positive and X negative examples, where X represents the threshold of χ^2 required to achieve the desired significance threshold.

Theorem 3 *Let R be any rule. Let P denote any parent of R generated by removing a single conjunct and D denote any specialization of R. Let p denote the significance threshold used to judge trivial specializations and let X be the smallest value such that* $P(\chi^2 > X) < p$. *If* $|E_+(P) - E_+(R)| \leq X$ *and* $|E_-(P) - E_-(R)| \leq X$, *then D is a trivial specialization.*

Proof:
Let Γ denote P's tests and t denote the single test in R missing from P. The antecedent of R is therefore $\Gamma \wedge t$. The antecedent of D, a specialization of R, is similarly $\Gamma \wedge t \wedge \Theta$ where Θ represents D's additional tests. For D to be a trivial specialization, it must fail the χ^2 test for one of its parents. We will show that D fails the χ^2 test for the parent Q whose antecedent is $\Gamma \wedge \Theta$.

The formula for Brute's χ^2 test for comparing Q and D is

$$\chi^2 = \frac{\left(|E_+(D)| - \mathcal{A}_\mathcal{D}(Q) \cdot |E(D)|\right)^2}{\mathcal{A}_\mathcal{D}(Q) \cdot |E(D)|} + \frac{\left(|E_-(D)| - (1 - \mathcal{A}_\mathcal{D}(Q)) \cdot |E(D)|\right)^2}{(1 - \mathcal{A}_\mathcal{D}(Q)) \cdot |E(D)|}.$$

If we rewrite this equation using $p = |E_+(D)|$, $n = |E_-(D)|$, $\Delta p = |E_+(Q) - E_+(D)|$ and $\Delta n = |E_-(Q) - E_-(D)|$, the formula simplifies to

$$\chi^2 = \frac{(n \cdot \Delta p - p \cdot \Delta n)^2}{(n + \Delta n)(p + \Delta p)(p + n)}.$$

The values of p and n are bounded above by $|E_+(R)|$ and $|E_-(R)|$ respectively since D is a specialization of R. An upper bound for Δp is obtained by comparing $\Delta p = E_+(Q) - E_+(D)$ and $E_+(P) - E_+(R)$. The set $E_+(Q) - E_+(D)$ contains all positive examples matching $\Gamma \wedge \neg t \wedge \Theta$ and the set $E_+(P) - E_+(R)$ contains all positive examples matching $\Gamma \wedge \neg t$. Since

$$\Gamma \wedge \neg t \wedge \Theta \Rightarrow \Gamma \wedge \neg t,$$

the set $E_+(P) - E_+(R)$ is a superset of $E_+(Q) - E_+(D)$. Therefore, the value of Δp is bounded by $|E_+(P) - E_+(R)|$. An identical argument bounds Δn by $|E_-(P) - E_-(R)|$.

The bounds on Δp and Δn make it possible to bound χ^2. The χ^2 curve can be split into three regions based on whether $n \cdot \Delta p$ is less than, equal to, or greater than $p \cdot \Delta n$. When $n \cdot \Delta p < p \cdot \Delta n$, the values of $\frac{\partial \chi^2}{\partial p}$ and $\frac{\partial \chi^2}{\partial \Delta n}$ are positive, and the values of $\frac{\partial \chi^2}{\partial n}$ and $\frac{\partial \chi^2}{\partial \Delta p}$ are negative.

These derivatives imply the maximum value of χ^2 for this region is when $p = |E_+(R)|$, $\Delta n = |E_-(P) - E_-(R)|$, $n = 0$, and $\Delta p = 0$. Substituting these values into the χ^2 formula and simplifying provides the following:

$$\max_{n \cdot \Delta p < p \cdot \Delta n} \chi^2 = |E_-(P) - E_-(R)|.$$

The same argument applies to the region where $n \cdot \Delta p > p \cdot \Delta n$, but with the signs reversed, and results in the following formula:

$$\max_{n \cdot \Delta p > p \cdot \Delta n} \chi^2 = |E_+(P) - E_+(R)|.$$

Since the value of χ^2 is 0 for the region where $n \cdot \Delta p = p \cdot \Delta n$, the maximum value for the χ^2 test comparing D and Q is therefore

$$\max \chi^2 = \max \Big(|E_-(P) - E_-(R)|, |E_+(P) - E_-(R)| \Big).$$

This guarantees D will be judged a trivial specialization whenever $|E_-(P) - E_-(R)| \leq X$ and $|E_+(P) - E_-(R)| \leq X$.□

Theorem 3 allows pruning rules that cover almost the same examples as one of its parents. While the theorem can be applied directly by comparing the examples matched by each rule to the examples matched by each of its parents, it is very expensive to determine the examples matched by each parent. The cost of evaluating a rule without an example cache is $O(LN)$, where L denotes the length of the rule Since example caches are not available, the cost of evaluating all parents is $O(L^2N)$. To avoid this additional complexity, Brute only applies Theorem 3 to the rule's parent in the search tree. Since the examples matching this parent is cached, the theorem can be applied with little overhead.

5.4.3 Empirical Analysis

We first consider the inductive performance of Brute, Gold-digger, and C4.5 for finding the ten best rules from our two Boeing manufacturing databases. Table 5.2 shows the results of running Brute, Gold-digger, and C4.5 on these two databases. Brute conducted a complete search on the Occupancy-time database and a depth-four search

on the Failed-parts database. The results are averaged over 50 random trials that split the data 50% for training and 50% for testing. The table shows the average accuracy and average coverage of the rules learned by each algorithm. Since Gold-digger and C4.5 often fail to learn ten rules, the table lists three different numbers for Brute's performance: Brute's performance on all ten rules, Brute's performance when restricted to learning the same number of rules as Gold-digger, and Brute's performance when learning the same number of rules as C4.5.

The performance of all three algorithms are similar on the Occupancy-time data. Brute found rules that averaged 1.0 percentage point higher than Gold-digger and 2.6 percentage points higher than C4.5. The coverage of the rules learned by Brute and Gold-digger are essentially identical, while C4.5 learned rules with almost twice the coverage of both algorithms. This large increase in coverage is likely to be more useful than the small increase in accuracy offered by Brute. On the other hand, C4.5 only extracts an average of 6.1 rules from this database. C4.5 is missing some useful rules since Brute finds ten rules with similar accuracy as the 6.1 rules found by C4.5. Gold-digger also finds ten useful rules.

Brute performs the best on the Failed-parts database. Brute found rules which averaged 18.8 percentage points higher than those found by Gold-digger and 28.1 percentage points higher than those found by C4.5. The coverage of Brute's rules were 6.4 percentage points higher than Gold-digger and 6.0 percentage points higher than C4.5. Furthermore, Brute always found a full complement of ten rules while Gold-digger could only find an average of 7.6 rules and C4.5 could only find an average of 1.4 rules. C4.5 failed to find any rules on 40% of the random trials. Table 5.2 shows the averages for the 60% of the seeds for which C4.5 found some rules.

Brute's running time on both databases is reasonable, taking 4:32 (min:secs) on the Occupancy-time database and 11:34 on the Failed-parts database. Brute searched to depth four to keep the search time on the Failed-parts database manageable. While Brute did not perform a complete search on this database, it still found substantially

Table 5.2: Comparison of Brute, Gold-digger, and C4.5 for finding ten interesting rules from the Boeing databases. The table shows the average accuracy and average coverage of the rules returned by each algorithm. Since Gold-digger and C4.5 do not always return ten rules, averages are shown for Brute learning the same number of rules as learned by Gold-digger and C4.5. While the three algorithms are comparable on the Occupancy-time database, Brute performs significantly better on the Failed-parts database.

Occupancy-Time Database

(Base rate = 11.1%)

Algorithm	Average Test Accuracy	Average Test Coverage	Rules Returned
Brute – *10 rules*	52.1	5.4	10.0
Gold-digger – *10 rules*	51.1	5.3	10.0
Brute – *# rules Gold-digger*	52.1	5.4	10.0
C4.5 – *10 rules*	51.8	9.1	6.1
Brute – *# rules C4.5*	54.4	5.5	6.1

Failed-Parts Database

(Base rate = 7.2%)

Algorithm	Average Test Accuracy	Average Test Coverage	Rules Returned
Brute† – *10 rules*	30.5	8.5	10.0
Gold-digger – *10 rules*	12.4	2.3	7.6
Brute – *# rules Gold-digger*	31.2	8.6	7.6
C4.5‡ – *10 rules*	1.2	0.5	1.4
Brute – *# rules C4.5*	29.3	6.5	1.4

† Searched to depth four.

‡ Failed to learn any rules on 40% of the random trials.

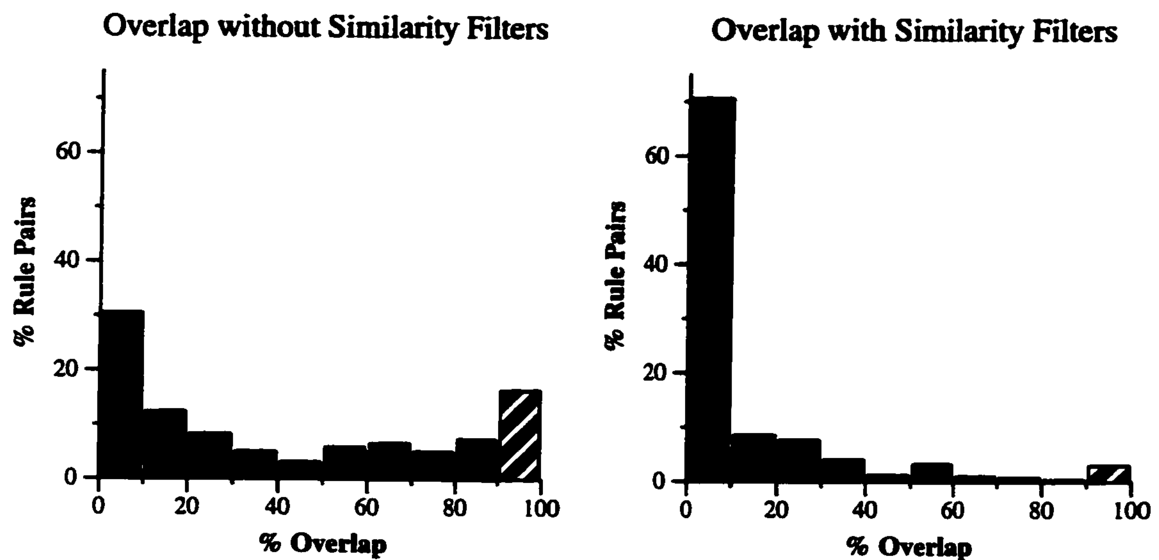
better rules than C4.5 and Gold-digger.

Another important question is the effectiveness of Brute's redundancy filters. Figure 5.9 shows two graphs for each database. The graphs on the left show the redundancy of Brute's rules in the previous experiments when not using most of its similarity filters. Correlation pruning was still used since the search is too expensive without it. The graphs on the right show the redundancy of Brute's rules in the previous experiments when using all its similarity filters. Each graph shows the percentage of ordered rule pairs with the indicated rule overlap. The overlap of two rules r_1 and r_2 is the percentage of examples matched by r_1 that are also matched by r_2. If r_1 matches a subset of the examples matched by r_2, then their overlap is 100%. If two rules share no examples, then their overlap is 0%. Rules with overlap above 90% are typically correlated and convey the same information. Correlated rule pairs are highlighted in the graph using diagonal stripes.

The graphs show that Brute's similarity filters substantially reduce redundancy. Without similarity filters on the Occupancy-time database, 14% of the rules returned by Brute have over 90% overlap. Brute's similarity filters reduce the percentage of rule pairs with 90% overlap to just 3%. Similar results occur on the Failed-parts database where the percentage of rules with 90% overlap drops from 21% to 10% with similarity pruning. Brute's similarity filters also reduce the percentage of other high-overlap rules including those with 70% and 80% overlap. On both databases when using Brute's similarity filters, over half the rule pairs have less than 5% overlap and are essentially non-overlapping. While Brute's similarity filters performed well on both databases, the 10% redundancy remaining on the Failed-parts data suggests there is room for further improvement.

Trivial-specialization pruning reduces Brute's search space by removing rules guaranteed to fail Brute's trivial-specialization filter. Table 5.3 analyzes the effectiveness of trivial-specialization pruning on the Boeing databases. The analysis shows that trivial-specialization pruning is effective and provides a search space reduction be-

Occupancy-Time Database



Failed-Parts Database

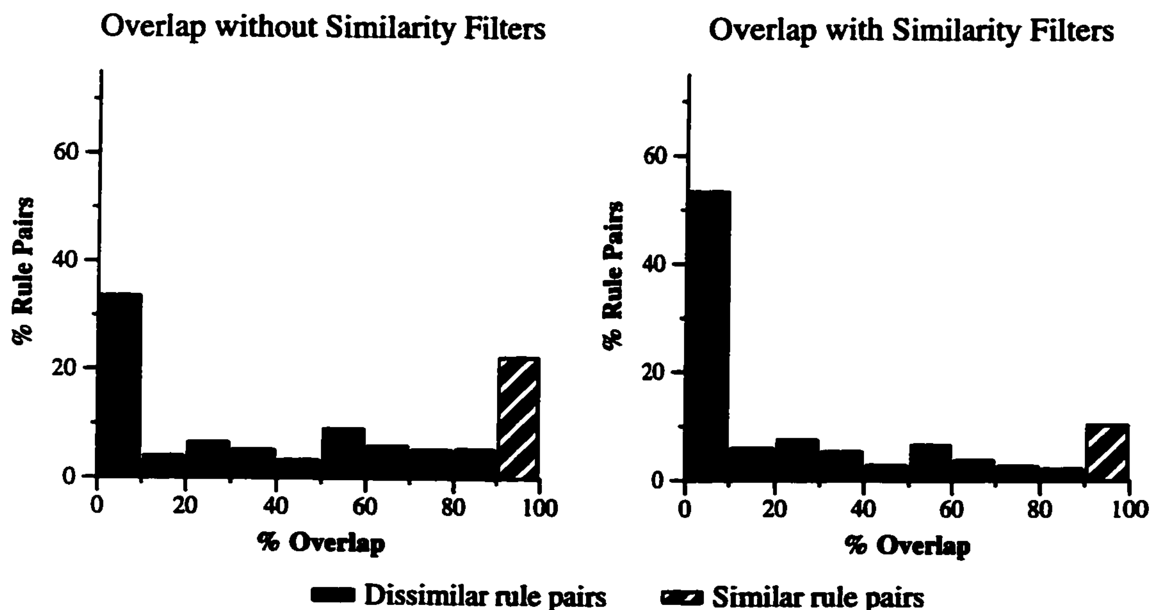


Figure 5.9: Effectiveness of Brute's similarity filters on the two Boeing databases. The graphs show the percentage of rule pairs returned by Brute with the specified rule overlap. Rule overlaps greater than 90% (diagonal stripes) indicate correlated rules. In both databases, Brute's similarity filters substantially reduce the percentage of rules with high overlap.

Table 5.3: The size of Brute's search space with and without trivial-specialization pruning.

Database	Rules Searched		Search Space Reduction
	Without Pruning	With Pruning	
Occupancy Time	37,422,967	5,341,745	7.0
Failed Parts	119,260,541	44,581,704	2.7

tween 2.7 and 7.0.

5.5 *Related Work*

ITRule was the first algorithm to apply massive search to data mining [Smyth& Goodman 91]. ITRule conducts a massive search to find the best M association rules. An association rule is similar to the if-then rules learned by Brute but allows any combination of tests to appear in the rule's consequent. ITRule learns a limited class of association rules where the consequent is restricted to a single test. ITRule's search algorithm only employs branch-and-bound pruning and therefore cannot perform the extensive searches performed by Brute. ITRule considers a smaller search space that does not include numerical attributes and only allows rules to contain a single test for each attribute. ITRule uses an information-theoretic evaluation function called the *J-measure* which makes a slightly different tradeoff between accuracy and coverage than that made by Laplace-depth. The J-measure does not include a complexity term and is likely to have difficulties with oversearching. While ITRule predates Brute, the two algorithms were developed independently.

Apriori [Agrawal *et al.* 93] also learns association rules but allows arbitrary conjunctions to appear in the consequent. Apriori returns all association rules meeting user-provided thresholds on accuracy and coverage. All associations are learned in

a two-step process. The first step finds all conjunctions with the required coverage, and the second step finds partitions of these conjunctions that form associations.

Apriori was recently extended to handle numerical attributes [Srikant&Agrawal 96]. The extended Apriori supports arbitrary numerical ranges and can automatically segment numerical attributes into subranges to improve efficiency. The extensions also include a mechanism for reducing the redundancy caused by numerical attributes. Apriori defines a *numerical specialization* of a rule to be any logical specialization of the rule generated by manipulating numerical thresholds. The numerical specializations of

$$\textit{If } \textit{hour} > 12 \wedge \textit{hour} < 17 \textit{ then } \textit{class} = \textit{reject},$$

include

$$\textit{If } \textit{hour} > 14 \wedge \textit{hour} < 17 \textit{ then } \textit{class} = \textit{reject},$$
$$\textit{If } \textit{hour} > 13 \wedge \textit{hour} < 15 \textit{ then } \textit{class} = \textit{reject},$$

but do not include

$$\textit{If } \textit{hour} > 13 \wedge \textit{hour} < 18 \textit{ then } \textit{class} = \textit{reject}, \tag{5.4}$$
$$\textit{If } \textit{hour} > 12 \wedge \textit{hour} < 17 \wedge \textit{month} = \textit{May} \textit{ then } \textit{class} = \textit{reject}. \tag{5.5}$$

Apriori defines a rule to be *R-interesting* if either its accuracy or its coverage differs by a factor of R from what would be expected after seeing its most-specific R-interesting generalization. Apriori filters out any rule that is not R-interesting. There are two key problems with R-interestingness. First, it fails to detect similarities between overlapping but highly-correlated rules, such as rule 5.4, that are not strict specializations. Second, it decides what differences are interesting using a fixed constant rather than statistical significance. This causes Apriori to judge some similar rules as distinct and some differing rules as similar. Apriori does not avoid trivial specializations or correlations.

RL [Provost *et al.* 93] also applies extensive search to data mining but uses a beam search to ensure tractability. Similar to Apriori, RL returns all rules which meet user-set thresholds on positive and negative coverage. RL does not include any mechanisms to avoid redundant rules.

OPUS [Webb 95] also learns rules using massive search, but it was not designed for data mining. While the algorithm has been extended to learn multiple rules using a greedy covering algorithm [Webb 93], this extension has the same inability to learn overlapping rules that plagues Gold-digger.

Gold-digger is similar to several algorithms for learning decision lists including AQ, Greedy3, and CN2. AQ [Michalski 69] was the first to employ Gold-digger's covering algorithm but used a different rule-learning strategy. AQ performs a beam search for 100% accurate rules but biases its search using a randomly chosen positive and negative example that must be correctly classified by every conjunct added. Greedy3 combines Gold-digger's covering algorithm with a true greedy search for 100% accurate rules and adds a postprocessing algorithm that builds a classifier from the rules selected. The original version of CN2 conducted a beam search for rules with maximum information gain. Information gain is not well suited to data mining because it considers purity across all values of the goal attribute rather than focusing on the value of interest. CN2 uses an information-theoretic variant of Brute's trivial-specialization filter to avoid learning overly-complex rules, but not for reducing redundancy. Since CN2 prunes subtrees of rules failing its trivial-specialization filter, it cannot learn rules that require adding non-informative conjuncts. Later versions of CN2 [Clark&Boswell 91] replaced information gain with Laplace accuracy but continued using the specialization filter.

5.6 Summary

The key difference between rule learning and data mining is data mining's requirement for learning multiple rules. The typical approach to data mining is to extract rules from the trees learned by decision-tree algorithms. However, we have shown that decision-tree algorithms are poorly suited to data mining because of biases inherent in their test-selection functions, their inability to learn overlapping rules, and their inability to make guarantees about the rules returned. Although we attempted to develop a greedy algorithm specifically designed for data mining, most of the same problems remained.

Conducting a massive search for the best M rules eliminates each of these difficulties but introduces the new problem of how to avoid redundant rules. We presented three complementary techniques for reducing rule redundancy: filtering trivial specializations, filtering correlated rules, and filtering numerically-similar rules. Combined, these techniques help prevent Brute from returning redundant rule sets. Trivial-specialization filtering also provides additional opportunities for pruning. Rules in a subtree will fail Brute's trivial-specialization filter whenever a rule differs from one of its parents by only a few examples. By pruning rules that meet this criterion, we substantially reduce the size of Brute's search space.

Brute was quite successful on the Boeing data-mining application, learning rules with substantially-higher accuracy and substantially-higher coverage than either Gold-digger or C4.5. Brute's success on the Boeing databases has made Brute the core component of Boeing's data-mining effort [Riddle *et al.* 95].

Chapter 6

CLASSIFICATION

Classification is the central problem in many machine-learning applications. The goal of classification is to identify a function that can correctly predict, for any example, the value of the goal attribute. An example of classification is identifying a function that can accurately predict whether a congressperson is either a Republican or Democrat from the congressperson's voting records.

Decision trees and decision lists are the two most common representations for prediction functions. As described in the previous chapter, decision trees are tree-structured classifiers where each node contains a single test and each leaf contains a value of the goal attribute. Examples are filtered down the tree according to each node's test and are assigned the class of the leaf node at the end of its path. A decision list is an ordered list of conjunctive rules. A decision list classifies examples by assigning to each example the class associated with the first conjunctive rule that matches the example.

While decision-tree algorithms are not easily amenable to massive search, most decision-list algorithms have a rule-learning component that can be replaced with Brute's massive-search algorithm. As a result, decision lists are ideal for extending Brute for classification.

This chapter proposes two algorithms for applying massive search to classification. Brute-greedy is an extension of the standard decision-list algorithm that uses Brute rather than greedy search for its rule-learning component. BruteDL in an extension of Brute-greedy that replaces its greedy covering algorithm with a novel method for forming decision lists [Segal&Etzioni 94]. The next section describes Brute-greedy,

and the following section describes BruteDL.

6.1 Brute-greedy

Though there are a variety of algorithms for learning decision lists, most have similar structure. Most algorithms combine a rule-learning component with a greedy covering algorithm. While most algorithms use the same covering algorithm, each uses a slightly different rule-learning component: AQ [Michalski 69] uses a beam search for 100%-accurate rules that cover a seed positive example and exclude a seed negative example, Greedy3 [Pagallo&Haussler 90] uses a greedy search for 100%-accurate rules, and CN2 [Clark&Boswell 91] uses a beam search for the rule that maximizes Laplace accuracy. Some of the algorithms add a postpruning step to reduce overfitting.

Figure 6.1 presents a generalized algorithm for learning decision lists. The algorithm consists of a greedy covering algorithm with a rule-learning component in its inner loop. The final decision list is passed through a postpruning filter to improve inductive performance. The generalized algorithm can be instantiated with specific rule-learning and postpruning algorithms to simulate most decision-list learning systems.

The generalized algorithm for learning decision lists suggests one approach for applying massive search to classification, instantiate the generalized algorithm's rule-learning component with Brute. The resulting algorithm, Brute-greedy, is essentially identical to CN2-ordered, but with massive search replacing beam search and Laplace-depth replacing Laplace accuracy. Brute-greedy does not use a postpruning function.[1]

Figure 6.2 compares Brute-greedy's performance to that of Greedy-greedy, the instantiation of the generalized algorithm with a greedy rule-learning component that maximizes Laplace accuracy. The performance of each algorithm was determined on fifty random splits of each database into 50% training data and 50% test data.

[1] Brute-greedy does not include CN2's significance test which was shown empirically to reduce CN2's inductive performance [Clark&Boswell 91].

```
FUNCTION LearnDecisionList(Tests, Database):RULESET
    TrainData = Database
    DecisionList = ∅
    REPEAT
        Rule = LearnRule(Tests, TrainData)
        AppendRule(DecisionList, Rule)
        TrainData = TrainData - MatchedExamples(Rule, TrainData)
    UNTIL Positives(TrainData) = ∅ OR Rule = EmptyRule(Goal)
    AddDefaultRule(DecisionList, TrainData)
    PruneDecisionList(DecisionList)
    RETURN DecisionList
END
```

Figure 6.1: Generalized algorithm for learning decision lists. Most existing algorithms are instantiations of this algorithm with specific rule learning (LearnRule()) and postpruning functions (PruneDecisionList()).

Brute-greedy finds classifiers whose test accuracy is almost universally better than the classifiers found using Greedy-greedy. Furthermore, when Greedy-greedy performed better, the improvement in accuracy is small as compared with the large improvements afforded by Brute-greedy. Brute-greedy performs exceptionally well on the Monk2 database which requires learning an exclusive-or relationship for which greedy-search algorithms are known to perform poorly.

Our results with Brute-greedy counter earlier results that suggest massive search reduces inductive performance of CN2-like algorithms [Webb 93, Quinlan&Cameron-Jones 95]. The key to Brute-greedy's success is its use of Laplace-depth to reduce overfitting. Early attempts using Laplace accuracy were prone to overfitting and therefore resulted in poor inductive performance.

While Brute-greedy improves CN2-like algorithms that use greedy search, it does not compare favorably to the best classification algorithms. Figure 6.3 compares the inductive performance of Brute-greedy to C4.5, a state-of-the-art decision-tree classifier. While Brute-greedy still shines on the Monk2 database, C4.5 performs

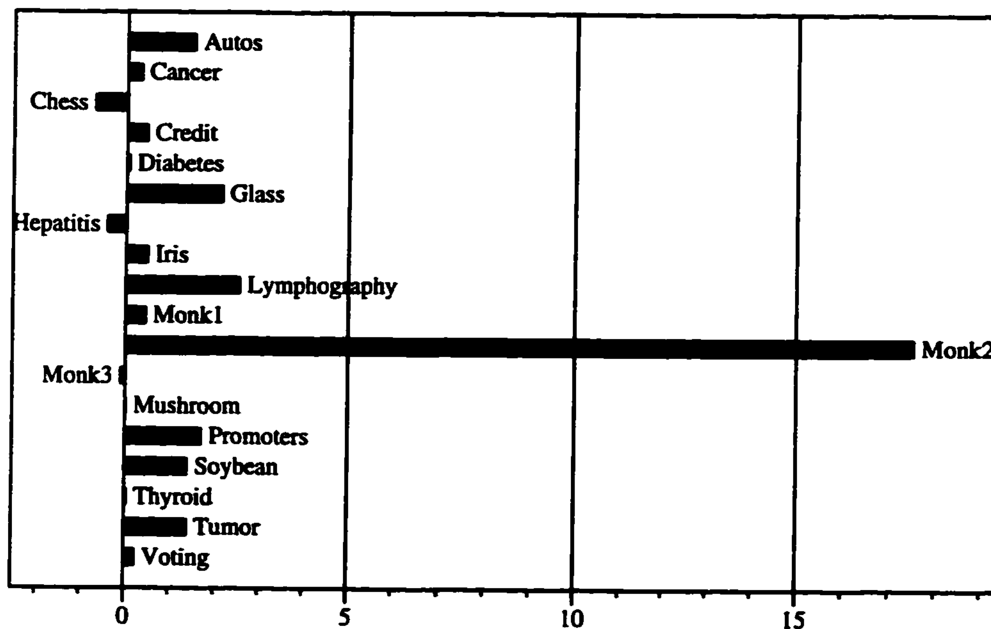


Figure 6.2: Comparison of the inductive performance of Brute-greedy and Greedy-greedy. Brute-greedy is the generalized decision-list algorithm instantiated with a massive-search rule-learning component. Greedy-greedy is the same algorithm but using greedy search. The graph shows that Brute-greedy outperforms Greedy-greedy on most databases.

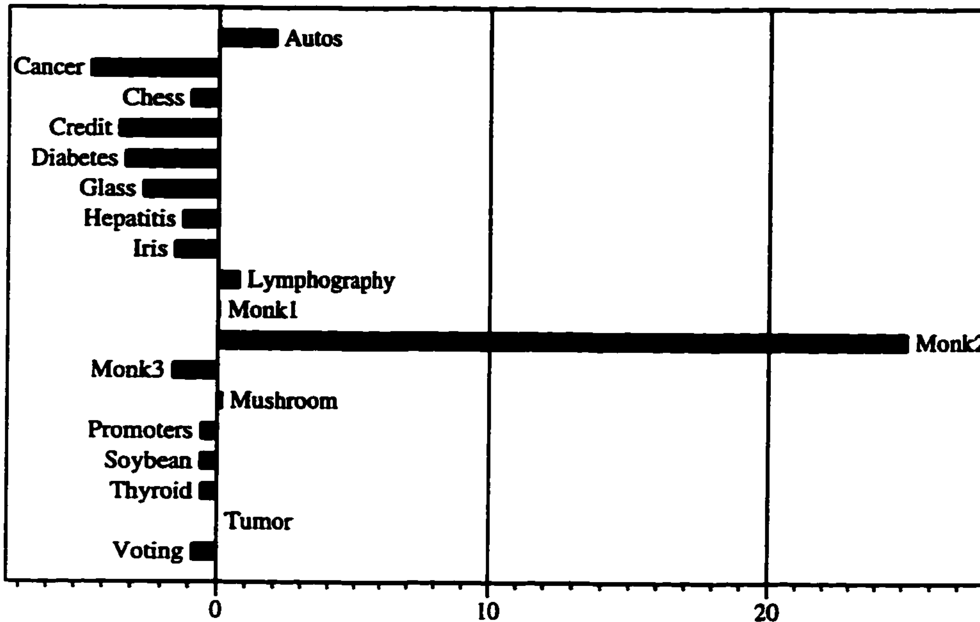


Figure 6.3: Comparison of the inductive performance of Brute-greedy and C4.5. While Brute-greedy improves on CN2-like algorithms using greedy search, it performs worse than the best classification algorithms.

better in all but three databases. As is discussed in the next section, the problem with Brute-greedy does not appear to be with massive search but with the control algorithm used to build its decision list.

6.2 BruteDL

Decision-list algorithms use greedy covering algorithms to avoid the *rule-overlap problem* — the accuracy of a decision list is *not* a straightforward function of the accuracy of its constituent rules. The accuracy of a decision list containing two rules r_1 and r_2, each having 80% accuracy and 50% coverage, depends critically on whether r_1 and r_2 overlap. The rules may not overlap at all, which yields a two-rule decision list with 80% accuracy and 100% coverage. However, the rules may have a 40% overlap

in which case the accuracy of the decision list (r_1, r_2) could decrease to 67% with a coverage of only 60%. In general, any algorithm that forms a classifier by combining rules learned separately has to overcome the rule-overlap problem.

Greedy covering algorithms address the rule-overlap problem by learning each successive rule from a training set where examples that match previously-learned rules are filtered out. This iterative approach is greedy — once the algorithm learns a rule, it is committed to keeping that rule in the decision list. All subsequent learning is based on this commitment.

While the greedy approach has proven effective in practice, it has several limitations. First, as pointed out by Clark and Boswell [1991], the interpretation of each rule is dependent on the rules that precede it. This makes decision lists difficult to comprehend because the learned rules cannot be considered in isolation. Second, on each iteration, fewer training examples are available for the learning algorithm, which hinders the algorithm's ability to learn. This is particularly important in situations where training data are scarce. Finally, poor rule choices at the beginning of the list can significantly reduce the accuracy of the decision list learned.

Nevertheless, Rivest showed that a greedy covering algorithm can provably PAC learn the concept class *k-DL*, decision lists composed of rules of length at most k [Rivest 87]. However, Rivest's PAC guarantee presupposes there exists 100%-accurate rules of length at most k that cover the training examples. This strong assumption neatly sidesteps the overlap problem because the accuracy of 100%-accurate rules remain unchanged regardless of the rules that precede them in the decision list. However, this assumption is often violated in practice. A full complement of 100% accurate rules of length at most k cannot be found when there is noise in the training data, when the concept to be learned is not in *k-DL*, or when the concept is probabilistic.

BruteDL is an extension of Brute-greedy that uses a new algorithm for solving the overlap problem. BruteDL's solution is based on the concept of *homogeneity* from

the philosophical literature [Salmon 84]. Informally, a homogeneous rule is one whose accuracy does not change with its position in the decision list.

Formally, let $\mathcal{I}$ denote the universe of examples. Let $\mathcal{T}$ denote the set of tests within a domain and G the set of goal classes. Let DL denote the set of all decision lists. Let $c(e)$ denote the correct classification of example e and $C(e,d)$ denote the classification that decision list d assigns to example e. To save space in the following discussion, we write rules as $A \rightarrow g$, where $A \subset \mathcal{T}$ and $g \in G$. When an example e passes all the tests in A, we say $e \in A$. Let $\mathcal{P}$ be a probability distribution over examples. We define the accuracy of a decision list d with respect to $\mathcal{P}$ as follows:

$$\mathcal{A}(d) = \sum_{\{e \in \mathcal{I} \mid c(e) = C(e,d)\}} \mathcal{P}(e).$$

We define the accuracy of a rule to be its accuracy on the examples that it covers:

$$a(A \rightarrow g) = \sum_{\{e \in A \mid c(e) = g\}} \mathcal{P}(e) \Big/ \sum_{e \in A} \mathcal{P}(e).$$

A *homogeneous rule* is a rule for which all specializations of the rule have the same accuracy as the rule itself. Formally, a homogeneous rule is a rule $A \rightarrow g$ such that, for all $B \subset \mathcal{T}$, the following holds:

$$a(A \wedge B \rightarrow g) = a(A \rightarrow g).$$

All 100%-accurate rules are homogeneous, but homogeneous rules need not be 100%-accurate. Thus, homogeneity can be viewed as a generalization of Rivest's solution to the overlap problem. This generalization is valuable in situations where concise 100%-accurate rules are not available.

BruteDL searches the space of conjunctive rules for maximally-accurate homogeneous rules and composes the rules found into a decision list. The remainder of this section is organized as follows. The next section introduces the theory underlying BruteDL. Section 6.2.2 explains the approximations to the theory used to make BruteDL practical. Section 6.2.3 then presents BruteDL's algorithm in detail. Finally, Section 6.2.4 compares BruteDL, Brute-greedy, and C4.5.

6.2.1 Homogeneous Decision Lists

This section describes the theory underlying BruteDL. Our goal is to demonstrate that the problem of finding a maximally-accurate decision list reduces to the problem of finding maximally-accurate homogeneous rules.

A homogeneous decision list is a decision list composed exclusively of homogeneous rules. We use *HDL* to refer to the set of all homogeneous decision lists. BruteDL learns only homogeneous decision lists. Does this restriction mean that in some cases BruteDL will be forced to learn an inferior decision list? In other words, are there cases where the best homogeneous decision list is less accurate than the best decision list? The answer is no. We say that two decision lists d and d' are *logically equivalent* if they classify all examples identically. That is, $\forall\ e \in \mathcal{I},\ C(e,d) = C(e,d')$.

Theorem 4 *For every decision list, there exists a homogeneous decision list that is logically equivalent.*

> **Proof sketch:**
> Let d be a non-homogeneous decision list and $r = A \rightarrow g$ be a non-homogeneous rule in d. We can replace r with a set of equivalent homogeneous rules. By performing this replacement for all non-homogeneous rules in d, we can find a logically-equivalent homogeneous decision list. Let t be any test not in A. The rule r can be replaced with the two rules $A \wedge t \rightarrow g$ and $A \wedge \neg t \rightarrow g$ without changing how d classifies examples. If these rules are homogeneous, we are done. If not, this procedure can be repeated until a set of homogeneous rules is found. A set of homogeneous rules must be found because there is a finite number of tests.□

Our solution to the overlap problem combines the notion of homogeneity with the intuition that the best rule to classify any example is the most accurate rule that covers the example. We define a *maximal cover* as a set of rules containing, for each example, the most accurate homogeneous rule that covers it. Formally, let $hr(e)$ be the set of homogeneous rules that match e. A maximal cover $M(\mathcal{I})$ of a universe of examples $\mathcal{I}$ is any set of homogeneous rules for which the following holds:

$$\forall\, e \in \mathcal{I},\ \exists\, r \in M(\mathcal{I}) \text{ such that } a(r) = \max_{r' \in hr(e)} a(r').$$

We now show that the problem of finding the maximally-accurate decision list reduces to the problem of finding a maximal cover for $\mathcal{I}$.

Theorem 5 *Any homogeneous decision list d whose rules form a maximum cover of $\mathcal{I}$ and is sorted by decreasing accuracy will have*

$$\mathcal{A}(d) = \max_{d' \in DL} \mathcal{A}(d').$$

Proof:

First we prove that d must be a maximally-accurate homogeneous decision list. Assume that $\mathcal{A}(d) \neq \max_{d' \in HDL} \mathcal{A}(d')$. There must exist a homogeneous decision list f such that $\mathcal{A}(f) > \mathcal{A}(d)$. Let e denote an example classified by a rule f_i in f and d_j in d such that $a(f_i) > a(d_j)$. Since $\mathcal{A}(f) > \mathcal{A}(d)$, such an example must exist. Since the rules of d form a maximum cover, d must contain a rule d_k such that $a(d_k) = \max_{r \in hr(e)} a(r)$. Furthermore, we have $a(d_k) \geq a(f_i)$ because $f_i \in hr(e)$. We have $k < j$ because $a(d_k) \geq a(f_i) > a(d_j)$ and d is sorted by decreasing accuracy. But if $k < j$, e should have been classified by d_k rather than d_j. This contradiction establishes that $\mathcal{A}(d) = \max_{d' \in HDL} \mathcal{A}(d')$.

Let g be a decision list such that $\mathcal{A}(g) = \max_{d' \in DL} \mathcal{A}(d')$. Assume $\mathcal{A}(g) > \mathcal{A}(d)$. By Theorem 4, there exists an $h \in HDL$ that is logically equivalent to g. We have $\mathcal{A}(h) = \mathcal{A}(g)$ because h and g are logically equivalent. But since h is homogeneous, we have $\mathcal{A}(d) \geq \mathcal{A}(h) = \mathcal{A}(g)$ which contradicts the assumption. Thus, $\mathcal{A}(d) = \max_{d' \in DL} \mathcal{A}(d')$.□

6.2.2 Implications

We now consider the implications of Theorem 5 for BruteDL. If BruteDL had access to the probability distribution $\mathcal{P}$ and the set of homogeneous rules, it would be straightforward to build an algorithm based on Theorem 5. In practice, BruteDL is given a set of training data from which it must approximate $\mathcal{P}$ and determine which rules are homogeneous.

BruteDL uses Laplace-depth as an approximation of the true accuracy of a rule. With an estimate for rule accuracy defined, it is possible to check whether a rule is homogeneous. The accuracy of a homogeneous rule should not change when adding conjuncts. Therefore, homogeneity can be checked by comparing the Laplace-depth of a rule with the Laplace-depth of all the rule's specializations. Since Laplace-depth is an approximation of the actual accuracy of a rule, a rule is considered homogeneous if all specializations have *roughly* the same Laplace-depth. We check for statistically significant differences in Laplace-depth using a χ^2 test.

Although not required by Theorem 5, it is desirable that the rules learned by BruteDL do not contain irrelevant conjuncts. An irrelevant conjunct is any conjunct whose presence does not affect the accuracy of a rule. We will call any rule with only relevant conjuncts *minimal*. Restricting BruteDL to minimal rules does not affect the class of concepts it can learn because, for every non-minimal homogeneous rule, there is a minimal rule with identical accuracy formed using a subset of the original rule's conjuncts. We check the minimality of rules using Brute's trivial-specialization filter.

6.2.3 Algorithm

BruteDL performs a massive search to find the best homogeneous rule that covers each example. Once a maximal cover has been found, the cover is sorted, and a default rule is appended. BruteDL uses the same search algorithm and pruning strategies as Brute, but rather than storing the single best rule, BruteDL records the best rule for each training example. BruteDL uses a novel data structure to efficiently track the best rule for each example. Like Brute, BruteDL stores a list of the current best rules in order of decreasing score. A new rule is the current best rule for some example if the new rule matches some example that is not matched by any of the stored rules with higher score. Brute checks this efficiently by storing with each of the current rules the set of examples matched by it and higher scoring rules. With

this representation, Brute can check if a new rule is the best for some example by finding the rule's position in the rule list and comparing the examples it matches to the examples stored with the next higher-scoring rule. If the new rule matches an example not stored with the next higher-scoring rule, then it is the best rule for that example and should be added to the rule set. Whenever a new rule is added, the set of examples matched by higher-scoring rules must be updated for any current rule of lower score. While performing the update, BruteDL removes any lower-scoring rule that is no longer the best rule for some example. The computational cost for updating the stored lists is minimal since updates are not frequently required.

Brute's branch-and-bound pruning must be extended to account for the examples covered by a rule. BruteDL's branch-and-bound pruning pretends to insert into the current rule set a rule that covers the same examples as the current rule but whose score is replaced by the subtree's upper bound. If this rule would not be incorporated into the current rule set by BruteDL's rule-storage algorithm, then no specializations of the rule could appear in the final decision list. BruteDL's branch-and-bound pruning removes any subtree whose root node fails this test.

Homogeneity is checked using a systematic search of all specializations of a rule. If a specialization is found with a difference in accuracy that is considered statistically significant, the homogeneity check fails. If no such specialization is found, the rule is deemed homogeneous. BruteDL's homogeneity check uses essentially the same search algorithm as Brute but does not use subsumption pruning because it is not applicable. BruteDL instead uses something similar that removes any specialization of a rule that matches the same examples as its parent. This is safe because, for every element of the pruned subtree, there is a corresponding rule in the parent's subtree with identical accuracy and coverage. Since the homogeneity check is searching for a single dissimilar rule, it is not necessary to search both subtrees.

BruteDL limits the cost of homogeneity checks by reducing their frequency. It is only necessary to check the homogeneity of a rule that is minimal and is the best

rule seen thus far for some example. If a rule does not meet these criteria, it cannot be part of the final decision list. Using this filter, BruteDL requires homogeneity checks for only a small fraction of the rules searched. Furthermore, the homogeneity checks for non-homogeneous rules are often inexpensive because the search can be terminated once a specialization with a significant difference in accuracy is found.

BruteDL forms its final decision list by sorting the maximum cover and appending a default rule. A default rule is necessary because the rules found by BruteDL, although required to cover the training examples, might not cover all test examples. BruteDL appends a default rule that predicts the most frequent class in the training data.

6.2.4 Experimental Results

Figure 6.4 shows the relative performance of BruteDL and Brute-greedy. We averaged the results for each experiment over 50 iterations, where each iteration split the available data into 50% for training and 50% for testing. BruteDL's performance is disappointing, it performs substantially worse than Brute-greedy on most databases. BruteDL's poor performance suggests a problem with either the assumptions underlying its theoretical model or with the approximations used to make it practical.

We experimented with various modifications of BruteDL to determine which assumptions and approximations were causing difficulty. While it is unclear which assumption or approximation it is breaking, we did discover a problem with how BruteDL selects a rule for each example.

BruteDL selects the highest-scoring rule covering each example. It makes this choice without considering whether the rule correctly classifies the example. This creates an odd situation where BruteDL can add a rule to cover an example, but the rule does not classify the example correctly. Since adding this type of rule lowers the decision list's training accuracy, it is also likely to lower the decision list's inductive performance.

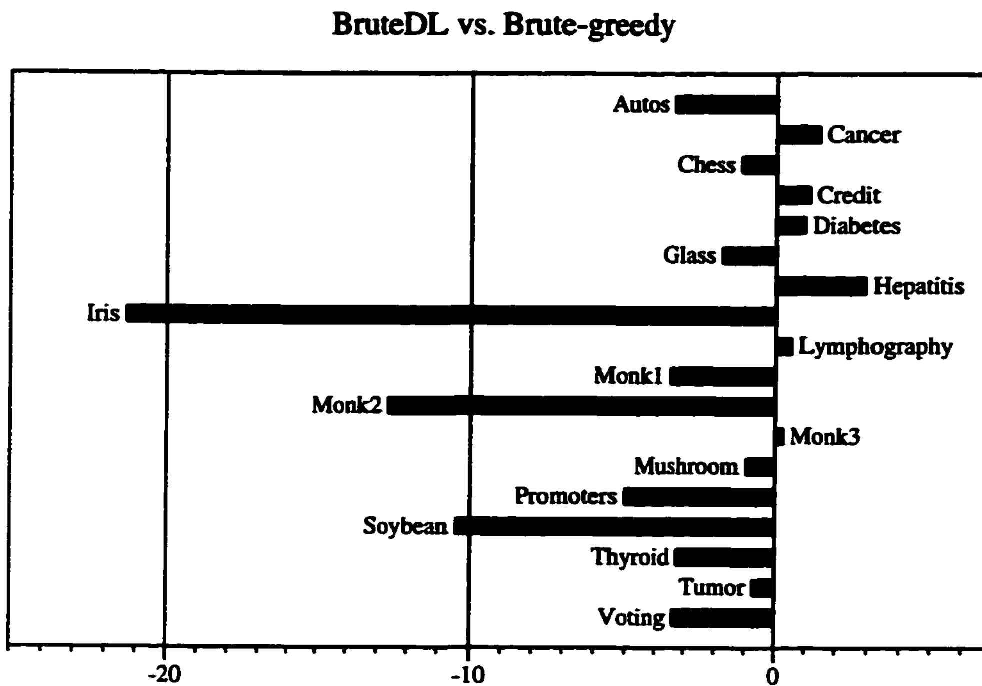


Figure 6.4: Comparison of BruteDL and Brute-greedy showing that pure BruteDL fails to improve inductive performance.

We can test this hypothesis by modifying BruteDL to select the highest-scoring rule for each example that classifies it correctly. Figure 6.5 shows that this modification almost universally improves inductive performance. Figure 6.6 shows that modified BruteDL performs on par with Brute-greedy. Modified BruteDL performs better than unmodified BruteDL on eight databases, worse on five databases, and about the same in the remaining five. Figure 6.7 shows that modified BruteDL performs better than C4.5 on a several databases. The large performance gains on these databases show the potential of BruteDL-style algorithms. However, C4.5's superior performance on most databases indicates more work is needed before massive search is the algorithm of choice for classification.

BruteDL is computationally more expensive than Brute because it must find the best rule for each example. Consequently, several of the databases were too large for

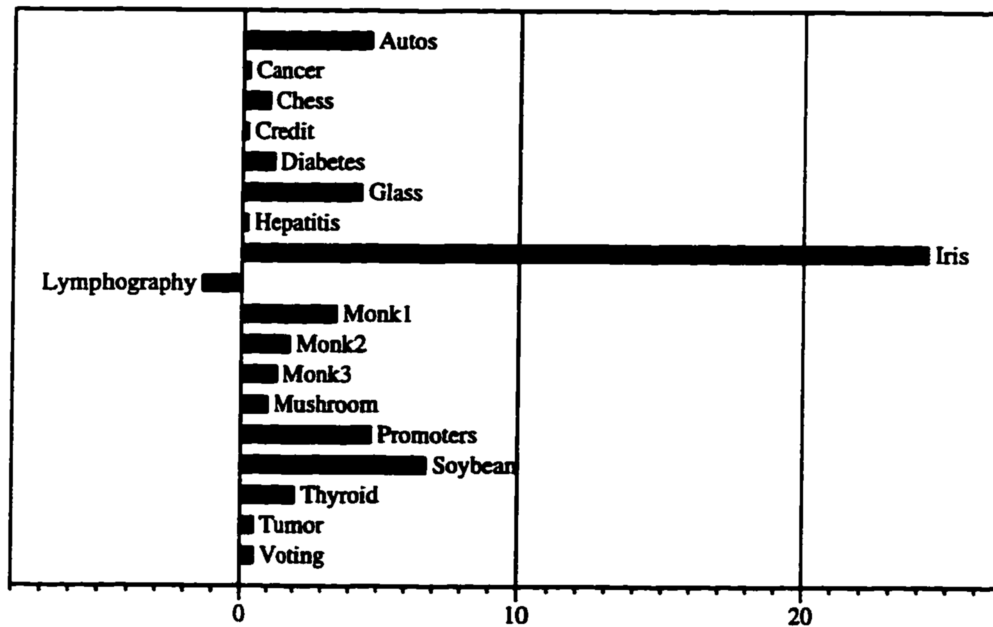


Figure 6.5: Comparison of the inductive performance of BruteDL and BruteDL modified such that the rule selected for each example classifies the example correctly. This modification almost universally improves performance.

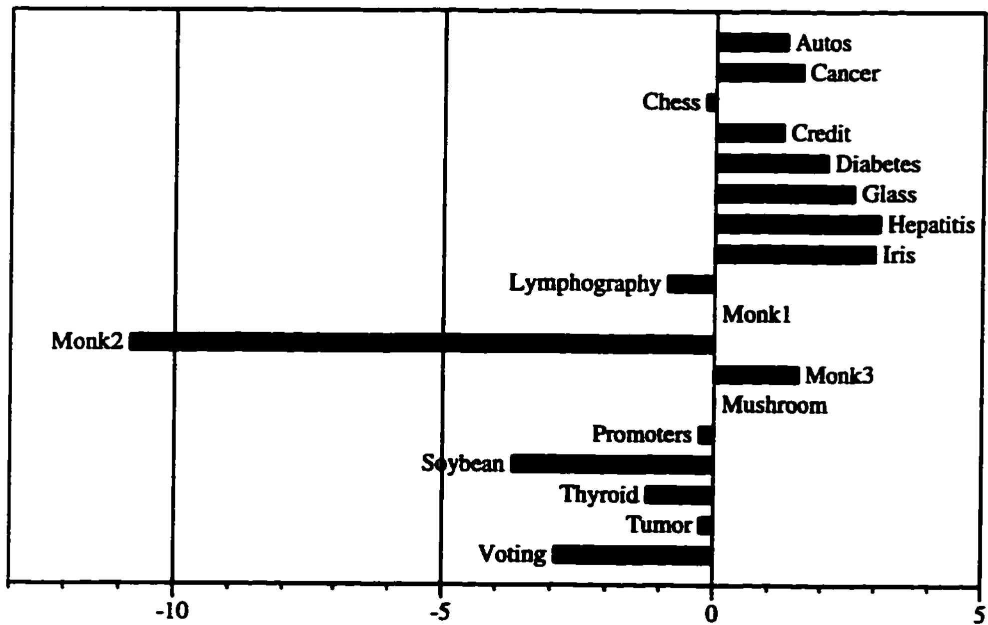


Figure 6.6: Comparison of the inductive performance of modified BruteDL and Brute-greedy. The two algorithms perform similarly, each outperforming the other in about half the databases.

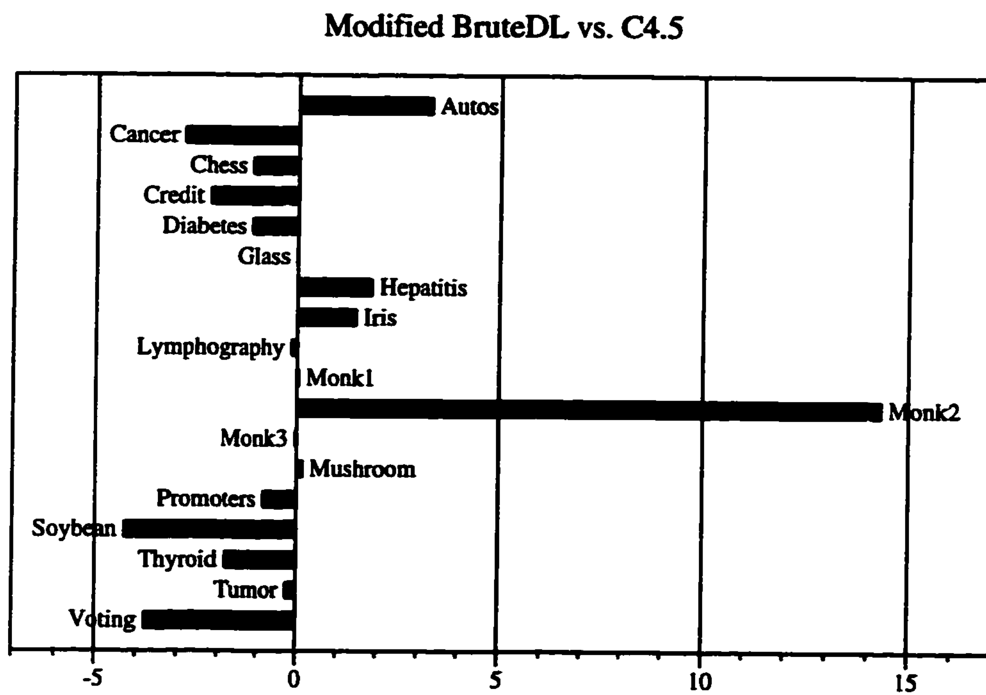


Figure 6.7: Comparison of the inductive performance of modified BruteDL and C4.5. While C4.5 performs better overall, the strong performance of BruteDL on three databases shows potential.

Table 6.1: Running times and search depths for BruteDL. CPU time is for a PowerPC-41T workstation.

Domain	CPU Time (min:sec)	Search Depth
Autos	5:30	3
Cancer	1:02	full
Chess	1:39	5
Credit	7:25	3
Diabetes	13:55	3
Glass	56:46	4
Hepatitis	20:54	5
Iris	0:10	full
Lymphography	25:07	full

Domain	CPU Time (min:sec)	Search Depth
Monk1	0:01	full
Monk2	0:04	full
Monk3	0:01	full
Mushroom	8:31	5
Promoters	31:04	5
Soybean	11:35	5
Thyroid	4:28	3
Tumor	0:10	full
Voting	0:01	full

a complete search. In each of these databases, a depth bound is used to limit search costs. Table 6.1 shows the execution times and depth bounds for each database.

6.2.5 *Critique*

Ideally, BruteDL's massive search would result in substantial improvements over greedy algorithms such as C4.5. Our experiments do not demonstrate this improvement for several reasons. First, on some data sets (e.g., Mushroom) we observe a ceiling effect — C4.5 is performing about as well as possible given the data set and attribute language. Second, in some cases, BruteDL overlooks homogeneous rules. BruteDL discards a rule as non-homogeneous when it has a specialization that differs significantly in accuracy from the rule itself. BruteDL performs a χ^2 test at $p = .005$ on each specialization of the rule to determine if its accuracy is significantly different. However, it is not the case that the probability that BruteDL incorrectly judges a rule to be non-homogeneous is .005. Although the probability that a single error is

.005, the probability that at least one of its N judgments is in error is $1 - .995^N$. The more specializations a rule has, the more likely it is to be incorrectly judged non-homogeneous.

On both the Voting and Cancer data sets, BruteDL incorrectly judged several key rules to be non-homogeneous. We can reduce the likelihood BruteDL will incorrectly judge a rule as non-homogeneous by using a lower p value for the χ^2 tests. By using $p = 0.00001$, BruteDL improves performance by 2.3 percentage points on the Voting database and by 4.6 percentage points on the Cancer database. However, simply increasing the confidence in individual χ^2 tests can cause BruteDL to treat a non-homogeneous rule as homogeneous. For instance, accuracy on the Monk2 data set decreases by 1.6 percentage points when we increase the confidence level. A more stable method of checking homogeneity is needed.

Finally, BruteDL's performance is limited on databases where it cannot search to sufficient depth to find accurate rules. For instance, 38.5% of the rules C4.5 finds on the Diabetes database were longer than BruteDL's depth bound. Heuristic search techniques (e.g., beam search) can be used when a pure depth-bounded search to the desired depth is too costly. The basic ideas behind BruteDL apply equally well to heuristic search.

6.3 Related Work

Brute-greedy descends from the AQ line of inductive algorithms that include AQ [Michalski 69], Greedy3 [Pagallo&Haussler 90], and CN2 [Clark&Niblett 89, Clark&Boswell 91]. These algorithms share Brute-greedy's iterative structure but use either a greedy or beam search to find the best rule. While Cover [Webb 93] extends CN2 to use massive search, it fails to improve performance because it lacks Brute-greedy's preference for short rules. The greedy covering algorithm shared by these systems limits their generalization ability because bad decisions made early in

the learning process adversely affect the learning of subsequent rules. Furthermore, greedy covering algorithms introduce rule dependencies that make their learned decision lists difficult to interpret. BruteDL's solution to the overlap problem avoids both these pitfalls by learning each rule in isolation.

Rivest [1987] describes an AQ-like algorithm for PAC learning the concept class *k-DL*, decision lists composed of rules of length at most k. Rivest's *k-DL* algorithm conducts a depth-bounded search of the space of conjunctive rules to find 100% accurate rules. *k-DL* repeats this depth-bounded search n times, where n is the number of rules in the learned decision list. We can improve on *k-DL* by restricting BruteDL to consider only 100% accurate rules. The homogeneity check can be dropped because 100% accurate rules are necessarily homogeneous. This restricted version of BruteDL will PAC learn *k-DL* using a *single* depth-bounded search of the space of conjunctive rules. The time complexity of the restricted BruteDL is asymptotically faster than *k-DL* by a factor of n. Furthermore, the unrestricted BruteDL is more general because it works for noisy domains, probabilistic concepts, and concepts not in *k-DL*.

PVM [Weiss *et al.* 90] does a massive search of the space of classifiers. PVM's search is not exhaustive because it uses several heuristics to reduce the search space. Even with heuristics, the doubly-exponential search space explored by PVM limits it to considering classifiers that are significantly smaller than those considered by either Brute-greedy or BruteDL.

Murphy and Pazzani [1994] used a depth-bounded search of the space of decision trees to analyze the relationship between the smallest decision tree and classification accuracy. A massively parallel Maspar computer and small databases were used to make a complete search of this doubly-exponential space possible. Our theory of homogeneity combined with Brute's efficient search algorithm significantly reduce the cost of depth-bounded search and make it practical for many databases.

Lin [1995] analyzed the performance of BruteDL's default-rule strategy. BruteDL

appends a default rule to its decision list to ensure it can classify all examples. Lin's work shows that, on small databases, BruteDL's default rule classifies most test examples. As a result, BruteDL's performance on small databases depends on the quality of the default rule. Lin suggests eliminating BruteDL's default rule and classifying examples not covered by the original decision list by dynamically generating new rules at classification time. The basic idea is, for each example not covered by existing rules, to use BruteDL to find the best homogeneous rule covering the example and then classify the example using the newly-found rule. Lin's results show that this method improves BruteDL's performance on small databases. For the databases used in this thesis, Lin's results only show a marked improvement on the Soybean database.

Many of BruteDL's features help to improve the human readability of its decision lists. As pointed out by Clark and Boswell [1991], the readability of a decision list suffers because the interpretation of each rule is dependent on the rules that precede it. BruteDL avoids this problem by finding homogeneous decision lists. Homogeneous decision lists are easier to understand because the interpretation of each rule is not dependent on its position. Furthermore, BruteDL attempts to include all relevant conjuncts within each rule while leaving out any irrelevant conjuncts.

6.4 Conclusion

This chapter introduces two new classification algorithms. Brute-greedy is an extension of CN2-like algorithms to use massive rather than greedy search. Unlike previous systems that extend CN2-like algorithms for massive search [Webb 93], Brute-greedy successfully improves inductive performance because its bias for short rules curbs overfitting. However, Brute-greedy's greedy covering algorithm limits performance.

BruteDL is a novel algorithm for learning decision lists that eliminates the need for a greedy covering algorithm. BruteDL conducts a single search for accurate ho-

mogeneous rules and builds a decision list from the rules found. We show that, in the limit, the problem of learning maximally-accurate decision lists reduces to the problem of learning maximally-accurate homogeneous rules. BruteDL's performance is on par with Brute-greedy's but is worse than the best classification algorithms such as C4.5. However, BruteDL's ability to outperform C4.5 on several databases suggests the potential of BruteDL's novel approach.

Chapter 7

CONCLUSION

7.1 *Summary*

Most machine-learning algorithms use greedy search because it is highly efficient and produces good results. However, greedy search often fails to find the best hypotheses because it only explores a fraction of the search space. In our experiments, the rules found by greedy search had nearly twice the predicted error as the best rules available. Massive search alleviates this problem by exploring large portions of the search space to find better hypotheses.

This thesis investigated massive search to answer three key questions: is massive search feasible? does it produce better solutions that greedy search? and in what range of tasks is massive search appropriate? We answered these questions by designing and implementing a massive-search algorithm for rule learning called Brute and by conducting experiments to ascertain its capabilities.

7.1.1 *Is massive search feasible?*

The naive algorithm for massive search is to enumerate all hypotheses, evaluate each hypothesis on the training data, and output the best rule found. Even with a fast rule-evaluation engine, the naive algorithm would take thousands of years to analyze most databases. Therefore, Brute's design must ensure the efficiency of massive search. Brute uses two techniques to reduce search costs. The first and most important technique, rule pruning, avoids searching portions of the search space that can be proven not to contain the best rules. The second technique, bit-vectors, evaluates

rules on thirty-two examples simultaneously and reduces rule-evaluation cost as much as fifteen times.

The pruning techniques appropriate for rule learning depend on the rule-evaluation function. Different rule-evaluation functions will require different pruning techniques. Rather than developing pruning strategies for a specific evaluation function, we defined the class *depth-monotone* evaluation functions and developed pruning strategies that work for any function in this class. Since most common evaluation functions are depth monotone, the rule strategies we present have wide applicability.

Brute's rule-pruning strategies include branch-and-bound pruning, subsumption pruning, and dynamic reorganization. Branch-and-bound pruning is a standard AI technique that computes an upper bound for each subtree and removes any subtree whose upper bound is below the current best rule. Branch-and-bound pruning applies to any search problem for which a suitable function exists for bounding the score of any subtree. By developing a bounding function for any depth-monotone evaluation function, we have shown that branch-and-bound pruning applies equally well to rule learning.

Subsumption pruning makes use of the monotonicity of depth-monotone evaluation functions. Monotonicity ensures that a rule's score increases with positive example coverage and decreases with negative example coverage. As a result, a subtree can be pruned if there exists a subsuming rule that covers a superset of its positive examples and a subset of its negative examples.

Dynamic reorganization is an improved method for maintaining systematicity that enhances the effect of other pruning rules. Brute ensures systematicity by imposing an ordering on the tests and only includes rules whose conjuncts adhere to the ordering. This ordering creates unbalanced trees where sibling rules higher in the ordering have significantly-smaller subtrees than those lower in the ordering. As a result of this imbalance, the number of rules removed by a pruning operation depends on the subtree's position in the ordering. Dynamic reorganization continuously adapts

the test ordering to assign pruned subtrees low rankings and therefore increases the number of rules pruned.

Brute reduces the cost of rule evaluation by representing examples using bit-vectors. Bit-vectors improve the typical linear encoding by allowing thirty-two examples to be processed simultaneously using the low-level bitwise-and operation available on most machines. While partitioning and sorting techniques reduce the algorithmic complexity of rule evaluation [Quinlan 86, Agrawal *et al.* 96], their large execution constants result in bit-vectors having a fifteen-fold advantage on many databases.

As a result of Brute's pruning techniques and fast rule-evaluation algorithm, Brute can *completely* search all of our benchmark databases in less than two minutes. To better challenge Brute's search algorithm, we tested Brute on the ten largest databases from the UCI data repository as well as on two Boeing manufacturing databases. Brute can completely search nine of these twelve databases in less than twenty-four hours, with most taking only a few minutes. Two of the databases which could not be completely searched can be searched to depths three and four respectively in the same twenty-four hour period. Brute could not analyze the final database because the memory needed for Brute's data structures exceeded the capacity of the test machine.

7.1.2 Does massive search learn better rules than greedy search?

Intuitively, massive search should learn better rules because its increased exploration of the search space is guaranteed to find higher-scoring rules. Higher-scoring rules should imply increased inductive performance if the evaluation function accurately captures the relationship between training-data performance and test-data performance.

However, the intuition that massive search should produce better rules is at odds with a long standing tenet of statistics and machine learning that evaluating large number of hypotheses reduces inductive performance. The difficulty is that the more

hypotheses that are considered, the better the chances of encountering a *fluke theory*, a theory that accidentally fits the data well but does not represent a real pattern. Quinlan and Cameron-Jones [1995] have dubbed this phenomenon *oversearching* and have demonstrated it empirically by comparing the inductive performance of varying amounts of search. While their results suggest that additional search decreases inductive performance, their results can also be explained by a faulty evaluation function that is not a good predictor of inductive performance.

In Chapter 4, we reproduced Quinlan and Cameron-Jones' experiments and analyzed the cases where limited search outperformed massive search. We found that the evaluation function they used, Laplace accuracy, tends to learn highly-complex rules that overfit the data. Overfitting is a well-known problem in statistics and machine learning in which an overly-complex concept is learned that too closely mimics the training data. The applicability of overfitting to oversearching is surprising because massive search learns general rules that cover large numbers of examples while overfitting usually results from learning overly-specific rules that cover a small fraction of the data. We explain this anomaly by introducing the concept of high-coverage overfitting and by showing that high-coverage overfitting is caused by the more familiar low-coverage overfitting.

Our explanation of oversearching implies that massive-search algorithms can benefit from standard techniques for reducing overfitting. We tested this hypothesis by designing a new evaluation function called *Laplace-depth* that reduces overfitting by extending Laplace accuracy to include a bias for short rules. While the new evaluation function only affords an improvement in half of our benchmark databases, the improvement it affords is much larger than the decrease it causes on the remaining databases. Laplace-depth learned rules that were two percentage points higher than Laplace accuracy on six databases while only decreasing performance by two percentage points on one database. Laplace-depth offers sufficient improvement for massive search to outperform greedy search on 13 of our 18 benchmark databases,

while performing equally well on the remaining five.

7.1.3 *In what range of tasks is massive search appropriate?*

After demonstrating that massive search is effective for rule learning, we focused on the question of what other machine-learning tasks is massive search appropriate. We demonstrated massive search's wide applicability by extending Brute to handle data mining and classification.

Data mining

Data mining is the extraction of useful information from large databases. Since the information extracted for data mining is usually represented as rules, data mining is an instance of the rule-learning problem already handled by Brute. However, data mining differs from simple rule learning because data mining requires extracting multiple rules from a single database. As a result, applying Brute to data mining requires extending its basic algorithm to learn multiple rules.

While Brute can learn multiple rules by returning the best M rules rather than the single best rule, this approach often returns rule sets where each rule is similar and conveys almost the same pattern. This problem occurs because similar rules tend to have similar scores. Avoiding this problem requires throwing out all but the best variant of each unique pattern.

Brute employs several techniques to detect similar rules. The first technique removes trivial specialization of good rules. A trivial specialization of a rule is any specialization that has essentially the same accuracy as the original rule. Trivial specializations can be formed by either adding a high-coverage conjunct that covers most of the original rule's examples or by adding a conjunct that is uncorrelated with the goal attribute. In either case, the result is a rule that is likely to have a similar score but conveys no additional information. Brute avoids returning trivial

specializations by filtering out any rule whose accuracy is statistically similar to one of its parents.

Brute's second technique for avoiding redundancy eliminates redundant rules caused by correlated tests. If two tests are highly correlated, then they can be used interchangeably without significantly affecting a rule's score. As a result, correlated attributes can result in many variations of the same pattern. Brute uses its subsumption pruning mechanism to reduce the number of correlated rules. Subsumption pruning helps because, whenever one rule subsumes another, there must be correlations among the rule's tests. However, subsumption pruning does not catch all correlations because Brute only applies subsumption pruning to sibling rules and because imperfect correlations do not always result in subsumed rules.

Brute's final technique removes redundancies introduced by numerical attributes. Tests for numerical attributes are of the form $Attr \leq T$ or $Attr > T$ where T is a numeric threshold. Numerical attributes introduce a large number of correlated tests because tests comparing similar thresholds are usually correlated. Since subsumption pruning does not catch all correlated rules, without any additional mechanisms, Brute would return many correlated rules involving numerical attributes. Brute avoids this problem by removing all but the best rule of all structurally-similar rules that differ only by the values of their numeric thresholds.

Brute differs substantially from the traditional data-mining technique of extracting rules from classifiers learned using decision-tree or decision-list algorithms. While neither decision-tree nor decision-list algorithms were designed for data mining, their popularity for data mining is a result of their good classification performance and their wide availability. Classification algorithms have two biases that can cause them to miss interesting rules. First, classification algorithms tend to favor globally-optimal rules at the cost of ignoring more interesting rules that have good local but poor global performance. Second, classification algorithms are limited to learning a set of mutually-exclusive rules whereas many interesting rules overlap. Brute avoids both

these biases by considering each rule independently and is therefore better suited to data-mining problems.

We compared Brute's data-mining performance to a decision-list algorithm we developed called Gold-digger [Riddle *et al.* 94] and a popular decision-tree algorithm called C4.5 [Quinlan 93]. We compared the three algorithms on two data-mining problems from a Boeing manufacturing domain. On the first database, the three algorithms performed similarly with Brute improving on Gold-digger by one percentage point and improving on C4.5 by 2.6 percentage points. On the second database, Brute found rules which averaged 18.8 percentage points higher than those found by Gold-digger and 28.1 percentage points higher than those found by C4.5. Furthermore, while Brute could find a full complement of ten useful rules from each database, Gold-digger could not find a full complement of rules on the second database, and C4.5 could not find a full complement on either database. Brute found more interesting patterns than either algorithm while maintaining high accuracy.

We also tested Brute's ability to reduce redundancy on these two data-mining tasks. On the first database, Brute's redundancy filters reduced the percentage of redundant rules from 14% to 3%. On the second database, the redundancy filters reduced the percentage of redundant rules from 21% to 10%. While both these databases show Brute's redundancy mechanisms perform well, the 10% redundancy remaining in the second database suggests that additional mechanisms are needed.

Classification

Decision-list algorithms combine a rule-learning algorithm with a covering algorithm to learn a list of rules that function as a classifier. Because these algorithms already have a rule-learning component, they can be easily extended to use massive search by replacing their greedy rule-learning algorithm with Brute. The resulting algorithm, Brute-greedy, outperforms the equivalent greedy algorithm on all but three of our eighteen benchmark databases.

While Brute-greedy outperforms other decision-list algorithms, its performance is not at the level of the best classification algorithms. For instance, C4.5 outperforms Brute-greedy on fifteen of eighteen databases. Brute-greedy's poor performance relative to C4.5 suggests that its performance is limited by its greedy covering algorithm.

Greedy covering algorithms are used in machine learning to eliminate the *rule-overlap* problem — the accuracy of a decision list is not a straightforward function of the accuracy of its constituent rules. Greedy covering algorithms avoid the rule-overlap problem by evaluating each rule in the context that it will appear in the final decision list. BruteDL uses a new covering algorithm based on a theoretical analysis of rule overlap. This analysis shows that rule overlap can be avoided by learning *homogeneous rules*. Homogeneous rules are rules whose accuracy is unaffected by the insertion of additional conjuncts. Homogeneous rules eliminate the rule-overlap problem since their accuracy is the same for all subsets of examples.

While theoretical results suggest that learning with homogeneous rules is ideal, the theory cannot be directly applied because it is impossible to determine unequivocally if a rule is homogeneous. BruteDL approximates the theory using statistical testing to learn rules that are likely to be homogeneous. The resulting algorithm outperforms C4.5 on four of eighteen databases but performs substantially worse on nine databases. While BruteDL improves on Brute-greedy's results, it is clear that more work is needed before massive search is the algorithm of choice for classification.

7.2 Future Work

7.2.1 Better Evaluation Functions

Oversearching is the biggest challenge facing massive search. The oversearching problem occurs when performing additional search reduces rather than increases inductive performance. Chapter 4 showed that the oversearching problem can be attributed to poorly-designed evaluation functions. If the evaluation function is not highly corre-

lated with rule accuracy, then oversearching will occur.

In Chapter 4, we took the first steps towards developing more robust evaluation functions by showing that the poor performance of Laplace-accuracy can be partially explained by high-coverage overfitting. Based on this analysis, we developed the Laplace-depth evaluation function which includes a preference for short rules that reduces high-coverage overfitting. Unfortunately, while Laplace-depth is an improvement, it does not eliminate oversearching.

If better evaluation functions could be found, the benefits of performing massive search would increase substantially. Furthermore, new evaluation functions will require a deeper understanding of the factors determining a rule's predictive ability. This understanding is likely to benefit all inductive algorithms, including those using greedy search.

7.2.2 Scalability

The second largest challenge for massive search is scalability: can massive search algorithms handle the gigabyte or even terabyte databases that are starting to become common in data-mining applications. Brute can handle databases with large numbers of training examples well since its running time grows linearly in this dimension. However, Brute's assumption that all training data can fit in main memory implicitly limits the database sizes it can handle. Redesigning Brute to handle databases larger than main memory is difficult.

Apriori [Agrawal *et al.* 96] handles large databases by leaving the database on disk and evaluating all rules at a given depth using a single scan of the database. The number of times Apriori reads the entire database from disk grows linearly with the maximum rule length. However, Apriori achieves its ability to handle large databases at the expense of storing all active rules in main memory. While this works well for small hypothesis spaces, it breaks down when evaluating billions of rules. A combination of the two approaches may allow massive-search algorithms to handle

large hypothesis spaces and large databases simultaneously.

Provost and Hennessy [1996] propose another approach that shows promise even though it was designed to solve a slightly different problem. Their algorithm finds all rules whose score is greater than a user-specified threshold. The algorithm works by breaking the database into S equally-sized segments that fit in main memory and running the search algorithm on each segment. Each iteration's threshold is set to ensure that all rules meeting the original threshold will be found in at least one of the searches. Rules meeting the global threshold can be found by re-evaluating the rules found by each iteration on the entire database. This algorithm scales well to large databases because the entire database is read from disk only twice. However, it is difficult to quantify the additional overhead required to perform S independent searches on a slightly lower threshold or how many rules must be compared against the entire database. Furthermore, it is unclear how to modify this algorithm to find the best M rules rather than all rules above a threshold.

7.2.3 Learning Classifiers

Brute-greedy and BruteDL's performance relative to C4.5 is disappointing. Brute-greedy's limitations appear to be the result of its covering algorithm and may benefit from newer covering methods [Cohen 95]. BruteDL, being a newer algorithm, has many avenues for possible improvement.

The first step to improve BruteDL is gaining a better understanding of the relationship between its underlying theory and the approximations used in its implementation. This may uncover additional insights as to why BruteDL's good theoretical underpinnings are not resulting in good inductive performance. At the very least, a better understanding is needed of why finding the best rule that classifies each example *correctly* performs better than finding the best rule that classifies each example.

The next step is to develop better methods for detecting homogeneity. The current method is computationally expensive and does not properly account for multiple

applications of its χ^2 test. Simply correcting the threshold used for the χ^2 test using Bonferroni adjustments [Feelders&Verkooijen 95] is likely to misjudge homogeneous rules because the accuracy of a rule's specializations is not independent. Sampling techniques are an interesting alternative because they may eliminate the independence problem while simultaneously reducing computational costs.

An alternative to both Brute-greedy and BruteDL style algorithms is directly searching the space of decision trees or decision lists. However, these spaces are difficult to search for two reasons. First, both spaces grow doubly exponentially in the size of the database description language. Second, both spaces are not easily amenable to branch-and-bound pruning. The difficulty is that partially-formed decision lists and decision trees do not make any commitments as to how each example will eventually be classified. This makes it difficult to bound classification accuracy. While both problems are challenging, new search-space organizations and pruning techniques may make massive search of classifier space feasible in the future.

7.2.4 First-Order Learning

First-order learning is the extraction of first-order rules from relational databases. Massive search can easily be applied to first-order learning because first-order learning is an instance of the general rule-learning problem. The main difference between first-order and relational learning is that the database for first-order learning contains logical relations rather than attribute-value vectors. This difference introduces both new opportunities and new challenges.

The main new opportunity results from the additional search-space complexity of first-order learning. Unlike propositional learning, it is possible to add useful conjuncts to a rule that do not restrict the examples covered by the rule. A conjunct that does not restrict coverage is still useful if it adds new variables that can be used by later conjuncts. However, conjuncts that do not restrict coverage cause problems with greedy search because there is nothing to distinguish which of several

nonrestrictive rules are best. Quinlan and Cameron-Jones [1993] solve this problem for the special case of *determinate literals* — literals that introduce new variables that have a one-to-one relationship with existing variables. However, current first-order algorithms cannot learn rules with nonrestrictive, nondeterminate literals whose relationship to existing variables is not functional. Massive-search algorithms could avoid this problem by adding each nonrestrictive, nondeterminate literal in turn and therefore should be effective on a wider range of problems.

While the potential advantages of massive search are greater for first-order learning, there are also at least two new challenges. The first new challenge is that the cost of evaluating first-order rules can vary dramatically from rule to rule with the computational cost of some rules growing exponentially with the number of free variables. This increased evaluation cost results in some individual rules being computationally intractable to evaluate. Furthermore, when computational resources are limited, variable evaluation costs make it necessary to choose between evaluating many inexpensive rules or a few expensive rules.

One solution to this problem that offers much promise is to use best-first search and include computational costs as part of the rule-evaluation function. This adds a bias to the search algorithm for evaluating low-cost rules first and explicitly encodes how to tradeoff increased accuracy and additional computational costs. The result is an algorithm that only evaluates expensive rules if the potential gain outweighs the evaluation costs. Besides offering computational advantages, this approach has some intuitive appeal. A variation of Occam's razor is that the best hypotheses are not only concise but easily computable. As a result, including computational cost is analogous to including rule complexity in the evaluation function for propositional rules and may also improve inductive performance.

A second new challenge arises from the common practice of making the closed-world assumption in first-order domains. Under the closed-world assumption, the positive examples are the set of tuples for which the goal predicate holds and all

other tuples are assumed negative. As a result, the total number of examples used for training is the total number of tuples that can be formed using the goal predicate's free variables. Since this grows exponentially with the number of free variables, the closed-world assumption often generates databases with millions or billions of examples. This is more examples than can be efficiently handled with current greedy algorithms let alone massive search.

FOIL [Quinlan 90] addresses this problem by selecting a random sample of all negative examples and using the sample for rule evaluation. However, the sample can be a poor estimator since it represents a tiny fraction of all negative examples. Zelle *et al.* [1995] also uses estimation techniques to avoid enumerating all negative examples. However, a variation of their algorithm may allow for exact calculation. The key idea is that the number of negative examples matching a rule under the closed-world assumption is simply the number of tuples matching the rule minus the number of positive tuples matching the rule. The total number of tuples matching a rule can be easily computed using standard matching algorithms when all variables are bound during the matching process. Any variables not bound can be efficiently maintained using *collection-oriented match* [Acharya&Tambe 92]. However, the advantages of this approach have yet to be tested empirically.

BIBLIOGRAPHY

[Acharya&Tambe 92] Acharya, A. and Tambe, M. Collection-oriented match: Scaling up the data in production systems. Technical Report CMU-CS-92-218, Carnegie Mellon University, 1992.

[Agrawal *et al.* 93] Agrawal, R., Imielinski, T., and Swami, A. Mining associations between sets of items in massive databases. In *ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, DC, May 1993.

[Agrawal *et al.* 96] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. I. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[Blumer *et al.* 87] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

[Breiman *et al.* 84] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees.* Wadsworth, 1984.

[Clark&Boswell 91] Clark, P. and Boswell, R. Rule induction with CN2: some recent improvements. In *Machine Learning - EWSL-91. Proceedings of the European Working Session on Learning.*, pages 151–163, Porto, Portugal, March 1991.

[Clark&Niblett 89] Clark, P. and Niblett, T. The CN2 induction algorithm. *Machine Learning*, 3(4):261–284, March 1989.

[Cohen 95] Cohen, W. W. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, Lake Tahoe, CA, 1995.

[Cormen *et al.* 90] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[Feelders&Verkooijen 95] Feelders, A. and Verkooijen, W. Which method learns the most from data? In *Preliminary papers of the Fifth International Workshop on Artificial Intelligence and Statistics*, pages 219–225, January 1995.

[Good 65] Good, I. J. *The Estimation of Probabilities: An Essay on Modern Bayesian Methods.* Research monograph 30. MIT Press, 1965.

[Korf 85] Korf, R. E. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Korf 92] Korf, R. Linear-space best-first search: Summary of results. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 533–538, July 1992.

[Lin 95] Lin, J. A comparative study of default strategies for a decision list learner. Master's thesis, Vanderbilt University, Nashville, Tennessee, December 1995.

[Michalski 69] Michalski, R. S. On the quasi-minimal solution of the general covering problem. In *Proceedings of the Fifth International Symposium on Information Processing*, pages 125–128, Bled, Yugoslavia, 1969.

[Mingers 89] Mingers, J. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4):319–342, 1989.

[Mitchell 82] Mitchell, T. Generalization as search. *Artificial Intelligence*, 18:203–226, March 1982.

[Murphy 94] Murphy, P. M. UCI repository of machine learning databases. [Machine-readable data repository]. Irvine, CA. University of California, Department of Information and Computer Science., 1994.

[Murphy&Pazzani 94] Murphy, P. M. and Pazzani, M. J. Exploring the decision forest: An empirical investigation of Occam's razor in decision tree induction. *Journal of Artificial Intelligence Research*, 1:257–275, 1994.

[Niblett 87] Niblett, T. Constructing decision trees in noisy domains. In *Progress in Machine Learning (Proceedings of the 2nd European Working Session on Learning)*, pages 67–78, Wilmslow, UK, 1987.

[Pagallo&Haussler 90] Pagallo, G. and Haussler, D. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–100, March 1990.

[Pazzani *et al.* 94] Pazzani, M. J., Merz, C., Murphy, P., Ali, K., Hume, T., and Brunk, C. Reducing misclassification costs. In *Proceedings of the 11th International Conference on Machine Learning*, pages 217–225. Morgan Kaufmann, 1994.

[Provost *et al.* 93] Provost, F. J., Buchanan, B. G., Clearwater, S. H., and Lee, Y. Machine learning in the service of exploratory science and engineering: A case study of the RL induction program. Technical Report ISL-93-6, Intelligent Systems Laboratory, Computer Science Department, University of Pittsburgh, Pittsburgh, PA, 1993.

[Provost&Hennessy 96] Provost, F. J. and Hennessy, D. N. Scaling up: Distributed machine learning with cooperation. In *Proceedings of the 1996 Workshop on Integrating Multiple Learned Models*, Portland, OR, August 1996.

[Quinlan 86] Quinlan, J. R. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Quinlan 87] Quinlan, J. R. Generating production rules from decision trees. In *Proceedings of IJCAI-87*, pages 304–307, 1987.

[Quinlan 90] Quinlan, J. R. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Quinlan 93] Quinlan, J. R. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, 1993.

[Quinlan&Cameron-Jones 93] Quinlan, J. R. and Cameron-Jones, R. M. FOIL: a midterm report. In *Proceedings of the European Conference on Machine Learning.* Springer Verlag, 1993.

[Quinlan&Cameron-Jones 95] Quinlan, J. R. and Cameron-Jones, R. M. Oversearching and layered search in empirical learning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1019–1024, Montreal, Canada, August 1995.

[Quinlan&Rivest 89] Quinlan, J. R. and Rivest, R. Inferring decision trees using the Minimum Description Length principle. *Information and Computation*, 80:227–248, 1989.

[Riddle *et al.* 92] Riddle, P., Etzioni, O., Pearson, C., and Segal, R. Process improvement through automated feedback (preliminary report). In *Proceedings of the Machine Learning Workshop on Integrated Learning in Real-World Domains*, July 1992.

[Riddle *et al.* 94] Riddle, P., Segal, R., and Etzioni, O. Representation design and brute-force induction in a Boeing manufacturing domain. *Applied Artificial Intelligence*, 8:125–147, 1994.

[Riddle *et al.* 95] Riddle, P., Fresnedo, R., and Newman, D. Framework for a generic knowledge discovery toolkit. In *Preliminary papers of the Fifth International Workshop on Artificial Intelligence and Statistics.*, Ft. Lauderdale, Florida, January 1995.

[Rissanen 78] Rissanen, J. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[Rissanen 86] Rissanen, J. Stochastic complexity and modeling. *Annals of Statistics*, 14(3):1080–1100, 1986.

[Rivest 87] Rivest, R. Learning decision trees. *Machine Learning*, 2:229–246, 1987.

[Salmon 84] Salmon, W. C. *Scientific Explanation and the Causal Structure of the World.* Princeton University Press, Princeton, NJ, 1984.

[Schlimmer 93] Schlimmer, J. C. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, MA, June 1993.

[Segal&Etzioni 94] Segal, R. and Etzioni, O. Learning decision lists using homogeneous rules. In *Proceedings of the 12th National Conference on Artificial Intelligence*, July 1994.

[Smyth&Goodman 91] Smyth, P. and Goodman, R. M. Rule induction using information theory. In *Knowledge Discovery in Databases*, pages 159–176. MIT Press, Cambridge, MA, 1991.

[Srikant&Agrawal 96] Srikant, R. and Agrawal, R. Mining quantitative association rules in large relational tables. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[Valiant 84] Valiant, L. G. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[Vapnik 95] Vapnik, V. N. *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, NY, 1995.

[Weaver&Germond 94] Weaver, D. L. and Germond, T., editors. *The SPARC architecture manual : version 9.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Webb 93] Webb, G. I. Systematic search for categorical attribute-value data-driven machine learning. In Foo, N. and Rowles, C., editors, *AI '93*. World Scientific, Singapore, 1993.

[Webb 95] Webb, G. I. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:431–465, June 1995.

[Weiss *et al.* 90] Weiss, S. M., Galen, R. S., and Tadepalli, P. V. Maximizing the predictive value of production rules. *Artificial Intelligence*, 45:47–71, September 1990.

[Zelle *et al.* 95] Zelle, J. M., Thompson, C. A., Califf, M. E., and Mooney, R. J. Inducing logic programs without explicit negative examples. In *Proceedings of the Fifth International Workshop on Inductive Logic Programming*, 1995.

VITA

Education

UNIVERSITY OF WASHINGTON
Ph.D., Computer Science and Engineering, 1997.

CARNEGIE MELLON UNIVERSITY
B.S., Mathematics/Computer Science, 1990

Publications

O. Etzioni, H. Levy, R. Segal, and C. Thekkath. The softbot approach to OS interfaces. *IEEE Software*, 12(4):42-51, July 1995.

R. Segal and O. Etzioni. Learning decision lists using homogeneous rules. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, July 1994.

P. Riddle, R. Segal, and O. Etzioni. Representation design and brute-force induction in a Boeing manufacturing domain. *Applied Artificial Intelligence*, 8:125-147, 1994.

O. Etzioni, N. Lesh, and R. Segal. Building softbots for UNIX (preliminary report). Technical Report 93-09-01, University of Washington, 1993.

P. Riddle, O. Etzioni, C. Pearson, and R. Segal. Process improvement through automated feedback (preliminary report). In *Proceedings of the Machine Learning Workshop on Integrated Learning in Real-World Domains*, July 1992.

O. Etzioni and R. Segal. Softbots as testbeds for machine learning. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, Menlo Park, CA, 1992.