Tools and Techniques for Decision Tree Learning

Tapio Elomaa

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland

Series of Publications A, Report A-1996-2
Ph. D. Thesis, Helsinki May 1996, vi + 116 + 24 pages
ISSN 1238-8645, ISBN 951-45-7389-7

Abstract

Decision tree learning is an important field of machine learning. In this study we examine both formal and practical aspects of decision tree learning. We aim at answering to two important needs: The need for better motivated decision tree learners and an environment facilitating experimentation with inductive learning algorithms. As results we obtain new practical tools and useful techniques for decision tree learning.

First, we derive the practical decision tree learner *Rank* based on the *Findmin* protocol of Ehrenfeucht and Haussler. The motivation for the changes introduced to the method comes from empirical experience, but we prove the correctness of the modifications in the probably approximately correct learning framework. The algorithm is enhanced by extending it to operate in the multiclass situations, making it capable of working within the incremental setting, and providing noise tolerance into it. Together these modifications entail practicability through a formal development process, which constitutes an important technique for decision tree learner design.

The other tool that comes out of this work is TELA, a general testbed for all inductive learners using attribute representation of data, not only for decision tree learners. This system guides and assists its user in taking new algoritms to his disposal, operating them in an easy fashion, designing and executing useful tests with the algorithms, and in interpreting the outcome of the tests. We present the design rationale, current composition, and future development directions of TELA. Moreover, we reflect on the experiences that have been gathered in the initial usage of the system.

The tools that come about are evaluated and validated in empirical tests over many real-world application domains. Several successful inductive algorithms are contrasted with the *Rank* algorithm in experiments that are carried out using TELA. These experiments let us evaluate the success of the new decision tree learner with respect to its established equivalents and validate the utility of the developed testbed. The tests prove successful in both respects: *Rank* attains the same overall level of prediction accuracy as C4.5, which is generally considered to be one of the best empirical decision tree learners, and TELA eases the execution of the experiments substantially.

Computing Reviews (1991) Categories and Subject Descriptors:

I.2.6 Learning [ARTIFICIAL INTELLIGENCE]: Concept learning, Induction, Knowledge acquisition

E.1 Data Structures [DATA]: Trees

F.2.2 Nonnumerical Algorithms and Problems [ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY]: Pattern matching

General Terms:
Learning, Algorithms

Additional Key Words and Phrases:
machine learning, inductive concept learning, decision trees, computational learning theory, empirical experiments

Acknowledgements

First and foremost, I thank my advisor Esko Ukkonen for his continued guidance and support of this project. Without him this dissertation would never have seen the light of day. I am, also, grateful to Esko for the opportunity to do research together with him.

Feedback from Pekka Orponen has been most helpful in finalizing this thesis. Heikki Mannila has had an off and on advisory role in my post-graduate studies. Also, Martti Tienari has shown interest to my progress. Kai Koskimies and Jukka Paakki initially introduced me to scientific work and set me to the path that led to this dissertation.

I am gravely indebted to all those that I have been fortunate enough to collaborate with during the years. In particular, I owe a great deal to those people that have contributed to the research that is reported in this dissertation. Section 3.4 extends work that was carried out together with Jyrki Kivinen. In addition, Jyrki has been my primary interlocutor on computational learning theory over the years. Niklas Holsti and Juho Rousu have participated in the development of the TELA system that is described in Chapter 4. Many others have also been involved in the project during its time.

I want to extend my gratitude to include all those (domestic and foreign) friends and colleagues that have made this time joyful and worth the effort. Just to mention a few of those that were not already acknowledged, let me thank David Aha, Henry Tirri, and Ian Sillitoe.

My parents, Karin and Pentti, and my darling Eeva deserve my sincerest gratitude: Their encouragement ultimately was the decisive factor that ever made me finish this work.

Finally, the financial support of the Academy of Finland, the Ministry of Education, the Chancellor of the University of Helsinki, the Dean of the Faculty of Science, and the Department of Computer Science is gratefully appreciated.

Contents

1 Introduction **1**

2 Preliminaries **9**
2.1 Top-down induction of decision trees . . . 9
2.2 Alternative approaches to decision tree learning . . . 16
2.3 Probably approximately correct learning . . . 19
2.4 Coping with random classification noise . . . 24

3 The Design of a Learning Algorithm **27**
3.1 The rank of a decision tree . . . 27
3.2 Finding consistent decision trees of minimum rank . . . 32
3.3 Incremental construction of minimum rank decision trees . . . 39
3.3.1 Updating a hypothesis efficiently . . . 39
3.3.2 The incremental algorithm . . . 43
3.3.3 On the number of erroneous predictions . . . 47
3.4 Learning decision trees in the presence of noise . . . 48
3.5 The algorithm Rank . . . 54

4 TELA—a Tool for Attribute-Based Induction **57**
4.1 Design rationale of TELA . . . 58
4.2 An overview of the system . . . 61
4.2.1 System architecture . . . 61
4.2.2 Experiment specification language . . . 64
4.2.3 Enclosing new algorithms in TELA . . . 66
4.3 Future development of TELA . . . 68
4.4 Related systems . . . 70
4.5 Practical experiences on using TELA . . . 71

5 Empirical Evaluation and Validation **73**
5.1 Experiment setting . . . 73
5.1.1 Experiment outline . . . 73

5.1.2 The algorithm implementations 76
5.1.3 Experiment domains 77
5.2 Empirical results 79
5.2.1 Three StatLog problems 79
5.2.2 Experiments in noise-free domains 83
5.2.3 Experiments in noisy domains 92
5.3 Summary and analysis of the results 98
5.4 Discussion 100

6 Conclusion 101
6.1 Summary 101
6.2 Remarks 103

A Proofs of Lemmas and Theorems 117

B Description of the Test Domains 123
B.1 Assessing credit card applications 123
B.2 Vehicle type identification 124
B.3 Diabetes prediction 125
B.4 Space shuttle radiator positioning 126
B.5 DNA sequence boundaries 127
B.6 The six-bit multiplexor function 128
B.7 LED digit identification 129
B.8 Chess endgame result 130
B.9 Primary tumor location 131
B.10 Soybean disease identification 132
B.11 Mushroom species classification 134

C Dynamic Interface for Rank 135

D Exact Measurements Under Different Noise Types 139

Chapter 1

Introduction

The study of *artificial intelligence* explores possibilities of making computers exhibit behavior that could be seen to be intelligent. The ability to learn is a characteristic feature of intelligent behavior. Hence, it is not surprising that *machine learning* has been a topic on the agenda from the very early days of artificial intelligence and computer science (e.g., [Wiener 1948, Turing 1950]).

Machine learning research is blooming once again mainly because of the following (almost) coinciding impetuses. First, building expert systems was commercially the most important application of artificial intelligence. However, the difficulty of eliciting knowledge from domain experts, or the *knowledge acquisition bottleneck* [Feigenbaum 1977], proved to make the construction of expert systems complicated, expensive, and time-consuming. Machine learning techniques were considered to be able to help to circumvent the problem. Second, *neural networks* [Rumelhart & McClelland 1986], or connectionist computational devices, reemerged as an important research topic after having once falsely been doomed computationally insufficient [Minsky & Papert 1969]. Learning is an important method of "programming" neural networks. Third, the introduction of the *probably approximately correct learning model* [Valiant 1984] made the theoretical questions of machine learning a popular research topic.

In machine learning research, as in studies of natural learning processes, there are many approaches and subfields. This thesis examines a simple, but demanding, learning model: *Inductive inference* from examples. In this model the *learner* is supplied with a *sample* of the *target concept*. Hence, this subfield is also known as *concept learning*. In machine learning the learner is a computer program that is called a *learning algorithm*. A concept is any subset of a domain known as the *instance space*. The elements of this domain are *instances* and, hence, a concept is simply a collection of instances. The sample consists of *preclassified instances*, or *(training) examples*. Thus, the learning scheme is *supervised*; i.e., we can consider that there exists a *teacher* supplying the class labels of instances. In the basic model the teacher is omniscient—it does not err on deciding the class labels. A *classification* is any finite partitioning of the instance space.

It is the learner's task to construct and output its *hypothesis* of the target concept. Here we are interested in situations where the hypothesis takes the form of a *classifier*—a total function assigning any element of the instance space into one of the classes.

In *attribute-based* machine learning [Kalkanis & Conroy 1991] the examples are vectors of attribute values. An *attribute* is any measurable relevant characteristic of the application domain; it can typically only have one of three types: Its value range may be *nominal* (unordered), *discrete* (ordered), or *continuous*. The classification of an example is given as a *class label*—a distinguished attribute. In the majority of cases the class attribute is only allowed to be nominal; however, it is also possible to have a numerical class attribute (e.g., [Boswell 1990b]). The task is to induce a classification procedure that can be used to predict the class label of further instances on the basis of their attribute values. For this to succeed, the learner must be able to *generalize* the information contained in the examples. Usually the prediction given by the classifier is *categorical*—a single class is definitely nominated—but *probabilistic* prediction is also feasible [Quinlan 1987a, 1990c].

Example. Let us think of a mail order company that wants to intensify its marketing efforts by identifying more refined customer profiles than those it has used hitherto. For that purpose a *learning algorithm* is employed. (Classical statistical techniques have, of course, long been used in marketing analysis and other similar tasks involving correlation determination, but several studies [Carter & Catlett 1987, Shepherd *et al*. 1988, Weiss & Kapouleas 1989, Feng *et al*. 1994, Michie *et al*. 1994] have indicated machine learning techniques to be superior even in these tasks.) The *target concepts* in this task are the different types of customers and the aim is to induce a description of a potential customer for a given product (*the hypothesis*). As the *sample* the company can use its records of customers and those people that have, in previous marketing campaigns, been identified as potential customers, but have not come through with an order (records of the latter constitute a set of *negative examples*). The *instance space* in this case ranges from the population of the World to, say, the male owners of British cars in Finland. The classification of an instance already exists in the records: Did the potential customer place an order (for a specific product) or not. Hence, the *class attribute* specifies which merchandise, if any, was purchased by the customer. Strict laws prohibit our enterprise from acquiring all the customer details it would desire. Something, though, can be gathered from the existing records—the gender of the customer, the method of payment, the characteristics of the customer's living surroundings, etc. Thus, as instance-describing *attributes* we can use, e.g., the binary attribute `sex`, the discrete-valued attribute `population`, which gives the size category of the customer candidate's place of residence, and the continuous-valued attribute `distance`, measuring the journey to the retail store closest to the customer's address. □

In supervised learning a faultlessly classifying teacher (natural classifier) is present.

Why, then, would we want to replace it with a mechanical classification procedure that is even liable to make mistakes? The possible reasons include, for example, the following. *Speed* of classification may be essential; a computer program is often much faster in its processing than a human or some other natural classifier. *Objectivity* may be at risk when a person processes (sensitive) data; computer programs do not inherently contain moral or other biases. Even if an omniscient natural classifier exists, it does not necessarily mean that the law governing the classification of instances is articulated. Machine learning techniques can be used for extracting a representation for the underlying natural law. This may involve simply *enhancement of comprehensibility*, since even human experts may have trouble expressing their knowledge intelligibly, *knowledge simplification*, which may bring new insight to the observed world, or, at the most ambitious level, *discovery of knowledge* that was previously unknown. *Processing massive amounts of data* is task often better entrusted to computers. There are many applications where large amounts of data are constantly generated and there are many existing collections of such data masses. Even if speed is not a primary objective the teacher may still collapse under the multitude of data.

In many potential application areas of machine learning a classification is naturally associated to instances. Consider, for example, any monitoring system in a production plant, say, which constantly produces instances (the state of the monitored system at a given time) together with an associated classification (e.g., ‘operational’ or the type of malfunction). As another example, consider a learning task where previously treated patients’ records are given to the learner as the sample with the intent to come up with a classifier that can be used to decide the treatment of future patients. Nevertheless, in general the assumption of an omniscient teacher is a very restrictive one; if machine learning was only applicable in cases where a real-world error-free classification procedure exists, its utility would be severely impaired. In fact, one of the underlying motivations in the development of machine learning has always been to “manage the unmanageable.” Moreover, even if a natural classifier exists, its application costs may be untolerable, so that in essence the situation is as if no teacher exists. For example, the correct treatment of some cancer patients can only be determined by surgical operation, which is exactly the action that one tries to avoid by using machine learning.

In cases, where a natural teacher is missing or is too expensive to use, an *unsupervised* learning algorithm, or a *clustering* algorithm [Duda & Hart 1973], can be put into use. In unsupervised learning it is left to the learning algorithm to discover the possible classes, or clusters, in which the instances might belong. Obviously unsupervised learning is not as effective as supervised learning. In this thesis we concentrate on supervised learning alone; it can be envisioned to constitute the high-level back-end part of full-scale learning systems of future.

Learning from examples alone is difficult; there must be some additional guidance given to the learner about which hypotheses it should prefer and where to search for them. Such underlying assistance is known as *inductive bias* [Utgoff 1986, Haussler 1988]. The bias of a learning algorithm cannot usually be attributed only to a single property of it; several details contribute to an algorithm's bias: The most obvious sources of bias are the *representation language of hypotheses* (linguistic bias) and the *heuristics* employed in the algorithm (algorithmic bias). The former guides the algorithm towards certain kinds of hypotheses and the latter prompts some search order for the hypotheses. A more explicit way to guide the induction process is to provide the learner with *background knowledge*. Then, in addition to the representation languages of examples and hypotheses, a third representation language is needed. This study focuses on learning algorithms that do not have access to background knowledge and use a *propositional* (variable-value propositions) hypothesis representation language.

In addition to the problems of missing and expensive teachers we, in practice, encounter teachers, that are not omniscient, and the problem of misinformation. In other words, in most practical classifier learning situations an element of random errors, or *noise*, is typically present. In theoretical studies one usually makes a distinction between *attribute noise* and *classification noise*, which affect the observed attribute values and the teacher's decision of an instance's class label, respectively. In the real world, no such clean cut can be made; in practice an unseparable combination of both noise types is predominant.

Any learning algorithm that is intended for serious use should provide for the effects of noise in the sample. Luckily, it is, in most cases, quite easy to take into account the possible errors in the training examples when designing a learning algorithm. One just has to adopt a statistical view to the sample: Instead of trusting each individual training example the learner gathers statistical evidence from the sample and trusts only observations that are backed up by a sufficient number of examples. Of course, this general principle is not always easy to implement in a learning algorithm. The algorithm developed in this work provably tolerates noise in the training data.

The most common approach to concept learning examines situations where all training examples are available from the outset and no further examples can affect the constructed classifier; it is subsequently only used to classification of new instances. This is known as *batch* learning. In the real world, however, it may be that examples are received (one at a time) over a span of time during which a classifier is already needed. In principle, one could, of course, store all previously received training examples, add the new one to the set when received, and rerun the batch learning algorithm for the extended sample. Not only would this waste effort, but it is not necessarily tolerated by the application. Consider, for example, an autonomous mobile robot with a concept learning component. If the robot, upon receiving a new example, has to abandon whatever task it was performing to rerun the—as the sample size grows—increasingly slow batch learner, it may loose sight of its actual goal.

The study of *incremental*, or *on-line*, learning aims at building learning programs that minimize the (average) amount of hypothesis reconstruction and, thus, computation time that is required for each new example. From the theoretical point of view, there is not much difference between incremental and batch learning. However, whether the training examples are received all at once or one at a time largely affects the design of a learning algorithm. The incremental learner is not aware of the ending of its training period, if ever. Therefore, it has to maintain a suitable hypothesis even at the intermediate stages. Since hypothesis updates have to be efficient, they must be based on local operations rather than global ones. This alone, in practice, makes on-line learning much more complicated than batch learning. We develop an incremental learner with the same time requirement as its off-line equivalent has.

What are the general evaluation criteria for hypotheses produced by learning algorithms? Obviously the single most important property of a classifier is its *classification accuracy*, the proportion of correct classifications among its predictions. In this study we do not consider situations where errors may have different severity. *Classification speed* is in many cases a crucial property that is requested from the hypothesis. For example, all classifiers having to process masses of data, naturally, have to be expedient. *Comprehensibility* of the resulting classifier is a central feature whenever a human operator gets to review it; any classifier that is not fully understood by an expert will be deemed unreliable and, hence, non-applicable. Lack of trust has had tragic consequences in the Three-Mile Island and Chernobyl nuclear power plant disasters.[1] The final generally posed requirement to a machine learning technique is a property of the learner rather than of its hypotheses: The learner must be *quick*. The time that the learning period is allowed to take varies relative to the characteristics of the application domain (e.g., the numbers of attributes, classes, and examples), but we want the learner to be asymptotically efficient—at most polynomial in the dominating parameters; often even more stringent requirements are posed to the asymptotic efficiency of a learning algorithm. Our learning algorithm will be only linear in the size of the sample.

The hypothesis has to be expressed in some way. One of the most successful representations has been the *decision tree* formalism. The success of decision trees can largely be attributed to the fact that they naturally meet three of the four requirements posed above—decision trees are very efficient to learn and to use and they are generally conceived as quite understandable. In addition, the learning process is flexible; it can easily be changed to produce different decision tree types (e.g., shallow [Núñez 1988] or linear [Arbab & Michie 1985]) according to the user's needs. Also, the comprehensibility of a decision tree is easily enhanced even more by converting it into a corresponding set of production rules [Quinlan 1987b, 1987c].

A decision tree is a recursive classification procedure that can be viewed as a tree

[1] Automatic control techniques in general were distrusted here rather than classifiers in particular. In these catastrophes an automatic control device correctly recommended shutdown, but its advice was ignored by the human operators because they lacked belief into the foundedness of the recommendation [Michie *et al*. 1994, p. 7].

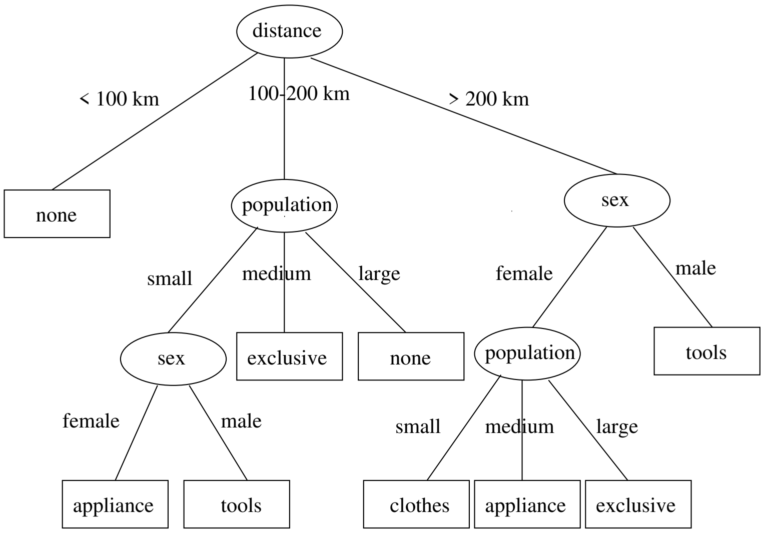


Figure 1.1: A decision tree describing the consumer habits of the clientele of a mail order enterprise.

with labeled nodes. A decision tree recursively partitions regions of the instance space into subregions. Each node corresponds to a region of the instance space. The root of the tree covers the entire instance space, its children divide the space into mutually exclusive regions, and the division process continues similarly all the way to the leaf nodes. Each leaf node has an associated class label, which is assigned by the classifier to any instance belonging to the corresponding region. Each internal node is associated with a test. In the simplest form a test only queries the value of one attribute. If a test T has C possible outcomes, the node associated with T has C children and the region covered by the node is partitioned into C subregions—one for each child.

Example. The hypothesis obtained by the mail order company of our previous example, when expressed as a decision tree, could look like that in Fig. 1.1. Note how the decision tree corresponds to a total function from the instance space (potential customers) to the set of classes (product groups): First the attribute `distance` partitions the instance space into three regions, two of which are partitioned further by the other attributes; any instance will be directed into one of the leaves by the tree. Interpreting the knowledge contents of the tree should be self-explanatory; for instance, the tree contains the information that (only) women living in small population areas over 200 kilometers away from the closest retail store tend to purchase clothing by mail order. One can also pay attention to secondary issues arising from this representation. For example, in what order do the attributes appear in the tree. The tree in Fig. 1.1 suggests that the distance from the closest retail store would contribute the most to the customers' consumer habits. □

This thesis develops tools and techniques for decision tree learning. Even though it is one of the most widely studied topics of machine learning research and notable successes have been gained, there are still many open questions to be solved in the underpinnings and practice of decision tree learning. One of the most salient deficiencies in the area is that theoretical results on decision tree learning are not up to the standard of the approach's practical success. Our first contribution is to take a theoretically sound decision tree learning algorithm, the *Findmin* of Ehrenfeucht and Haussler [1989], and evolve it into a practical tool.

The formal results of this thesis answer to the practical demands presented above: The decision tree learning algorithm that we develop is able to *manage multivalued attribute and class ranges*, thus retaining the intelligibility of the resulting decision tree; in it *incremental learning is supported*, making the algorithm applicable to wide range of problems; finally, it is endowed with the *capability to handle noise*, which is an unsurmountable problem in the real world. Let us emphasize that all these improvements are developed within the formal learning framework of Valiant [1984]; i.e., all modifications lead to provably good behavior. In addition to the individual analytical contributions, the design process in itself is a novelty; hitherto the theoretical design process of learning algorithms has attracted only some interest fairly recently [Maass 1994, Auer *et al.* 1995].

The developed algorithm's performance is compared with the best empirical decision tree learning algorithms in real-world learning tasks. This part of the work can be seen as a direct continuation for the empirical work of Michie *et al.* [1994]. It is of utmost importance to validate the performance of an algorithm on a wide variety of application domains in order to obtain a good overview of its general utility. It is the easiest task to come up with a good learner for a particular task, but a tool that is more generally useful defies solution. Furthermore, we do not content ourselves with simply comparing the prediction accuracies of the test programs, like Michie *et al.* [1994] did, but take a wider perspective by observing differences along the other quality measurements as well.

The other constructive contribution of this work is an environment facilitating experimentation and comparison of attribute-based learning algorithms. There are many learning methods easily available for the interested user, but substantial effort has to be offered to operate them. Therefore, there is a strong demand for a platform that could assist the user to easily operate many diverse learning tools. Moreover, fair and unbiased algorithm comparisons are impossible without a common control environment. We present the design rationale and an overview of TELA—a publicly available environment that has been implemented for these purposes.

The material is organized as follows. First, Chapter 2 recapitulates the basics of decision tree learning and computational learning theory, which are preliminaries for the rest of this dissertation. The first part reviews common practical decision tree learning techniques and presents a brief overview of less often used approaches to decision tree learning. In the second part Natarajan's [1991] modification of Valiant's [1984] formal learning framework is recapitulated. This modification deals with function class learning situations. The model is developed further by employing Sakakibara's [1993] ideas of coping with random errors in this extended model.

Chapter 3 develops a practical learning algorithm out of the decision tree learning protocol put forward by Ehrenfeucht and Haussler [1989]. Three major conceptual modifications are presented and several practical improvements are implemented in the new algorithm named *Rank*. First, the method is changed to handle multivalued variables and classes instead of dealing with binary values only, like the original method does. Next, we show how an efficient incremental variant of the learning algorithm can be developed. Finally, Ehrenfeucht and Haussler's method is extended to cope with situations that are affected by classification noise. At each stage we ensure that the formal learnability properties of the original algorithm are preserved: Each modification leads to a polynomial-time algorithm that is guaranteed to produce a decision tree with desired properties. Theoretical results play an important part in this chapter.

In Chapter 4 we introduce TELA—a generic platform for testing attribute-based learning algorithms. Testing and comparing different implementations and techniques on sample data is an integral part of the design and application of inductive machine learning programs. It involves many simple, but unavoidable auxiliary tasks that are usually not supported by the learning tools themselves. TELA has been developed to alleviate these in such a manner that the worries and vexations of the user are minimized. The algorithm *Rank* has been incorporated into the TELA platform.

Chapter 5 applies the algorithm *Rank* into many problems arising from real world. A series of tests, consisting of varying kinds of experiments, is reported and the results are analyzed meticulously. These tests are intended to empirically evaluate *Rank*'s utility with respect to other, more established inductive learners. At the same time we want to validate the TELA environment by showing how a long series of tests can be run under it without any complications.

Finally, Chapter 6 concludes the dissertation by summarizing the work and by discussing some issues raised by the work reported in the earlier chapters. In particular, we consider future research directions that could be taken to carry further the work that is reported in this dissertation.

Chapter 2

Preliminaries

This chapter surveys decision tree learning and computational learning theory, both of which are prerequisites for understanding the subsequent chapters. Section 2.1 recapitulates the most frequently used approach to decision tree learning. Alternative ways of learning decision trees are briefly surveyed in Section 2.2. The basic theoretical model of concept learning is introduced in Section 2.3 and Section 2.4, then, discusses how it needs to be changed for modeling noise-affected situations.

2.1 Top-down induction of decision trees

The literature on decision tree classifier learning is voluminous and constantly growing. We do not try to survey all aspects of it here; the interested reader is referred to consult one of the extensive survey articles, like the ones by Quinlan [1990a] and Safavian and Landgrebe [1991]. Also, an excellent survey of current topics on decision tree learning and closely related areas appears in the manual for the IND system [Buntine & Caruana 1993]. Instead of covering the research trends extensively, we concentrate on the common core of the approaches: We present the generic tree growing methodology and discuss only the most important variants of it. For an overview of inductive learning in general many sources exist (e.g., [Carbonell *et al.* 1983, Kalkanis & Conroy 1991, Langley 1996]).

The basic approach to decision tree learning is a heuristic hill-climbing search without backtracking or look-ahead. This is known as *top-down induction of decision trees* (TDIDT) [Quinlan 1986b]. TDIDT is typically performed in two steps—*growing* and *pruning*. In the first step a decision tree corresponding as closely as possible to the training data is constructed greedily, starting from the root, by heuristically selecting attributes to be tested at the nodes of the tree. It has been observed that a decision tree that fits the training data too closely will be a poor predictor of the class label of further instances. Hence, the tree needs to be pruned back in the second step in order to reduce its

Procedure 2.1 *StoppingCondition*(S, C)
input: a set of examples S.
output: a truth value. (Out parameter C records a class name.)
begin
(1) **if** all examples in S have the same class C **then return true else return false fi**
end.

Procedure 2.2 *MakeTree*$(a, T_1, \ldots, T_{|a|})$
input: an attribute a and a list of $|a|$ decision trees, where $|a|$ is the cardinality of a's range.
output: a decision tree.
begin
(1) Construct and return a decision tree such that attribute a is the label of its root and
(2) that trees $T_1, \ldots, T_{|a|}$ are its subtrees (T_i is joined to the root by an edge labeled by i)
end.

Procedure 2.3 *GrowTree*(A, S)
input: a set of examples S on attributes A.
output: a decision tree that is consistent with S.
begin
(1) **if** *StoppingCondition*(S, C) **then return** a leaf node labeled by C
    else
(2)   *Select* an attribute a from A;
(3)   Partition S into subsets $S_1, S_2, \ldots, S_{|a|}$ so that all examples having value i for
(4)   attribute a are assigned to subset S_i, where $|a|$ is the cardinality of a's range;
(5)   **for each** $i \in \{1, \ldots, |a|\}$ **do** $T_i \leftarrow$ *GrowTree*$(A \setminus \{a\}, S_i)$ **od**;
(6) $T \leftarrow$ *MakeTree*$(a, T_1, \ldots, T_{|a|})$;
(7)   **return** T
    fi
end.

dependence on the training examples. Also, pruning lets the learning algorithm tolerate the effects of noise.

The tree growing phase basically takes the form of Procedure 2.3 *GrowTree*. The procedure chooses an attribute from the set of available ones (Line 2), divides the examples belonging to the subsample under consideration into subsets according to their value of the chosen attribute (Lines 3–4), and recursively grows a decision tree for these sets (Line 5). The division process terminates when all examples belonging to the subset under consideration are of the same class (Line 1). The subprograms *StoppingCondition* and *MakeTree* have been segregated from the actual growing procedure for subsequent purposes.

Procedure *GrowTree* produces a decision tree that is consistent with the sample. All consistent decision trees are not considered equally good. It is generally accepted that simplicity should be favored in the classifiers produced by inductive learning programs. Reasons for favoring small classifiers are many: According to Occam's Razor [Blumer *et al*. 1987] a simple explanation is more likely to be correct than a complex one, the average cost of classifying instances by a simple classifier is smaller than that when using a complex classifier [Núñez 1988, Tan & Schlimmer 1990], and the relevant things (attributes, cut points, etc.) will be more clearly visible in a simple rule (comprehensibility). However, because of the inherent trade-off between simplicity and accuracy [Fisher & Schlimmer 1988, Iba *et al*. 1988, Bohanec & Bratko 1994] one is doomed to balance between these desired properties. Learning decision trees that are optimal with respect to many criteria, unfortunately, turns out to be a NP-complete problem [Hyafil & Rivest 1976, Comer & Sethi 1977, Murphy & McCraw 1991, Hancock *et al*. 1995] and, hence, intractable in practice.

Procedure *GrowTree* still leaves many details of decision tree growing unspecified. Most conspicuously it does not fix any method of choosing an attribute from among the available ones; for that end one commonly employs an information theoretic evaluation function to merit the candidate attributes. It is used to assess which division by a candidate attribute gives the best subsamples for predicting the class labels of training examples. More precisely, the function is used to approximate, for each candidate attribute a in A, the increase of information ΔI (about the class labels of training examples) that would be gained by dividing the subsample S under consideration into subsets $S_1, \ldots, S_{|a|}$ according to the value of the attribute a when compared to the situation where S is left undivided. The heuristic implements some form of inductive bias. The common goal of these measures is to keep the resulting tree as concise as possible without sacrificing much of its accuracy. The most generally used information theoretic evaluation functions belong to the family of *impurity measures* [Breiman *et al*. 1984]. For a thorough account of attribute selection criteria and their background see Kononenko and Bratko [1991] and for empirical comparisons of different heuristics see, e.g., Mingers [1989a] or Buntine and Niblett [1992].

Example. Quinlan's [1983, 1986b] ID3 algorithm uses the information theoretic concept *mutual information*, or *information gain* function, as its attribute merit function; it can be formalized as follows.

Let C be a discrete random variable with range R_C. For any $c \in R_C$ let $p(c)$ denote the probability $P\{C = c\}$. The (Shannon) *entropy* of C, $H(C)$, measures the information provided by an observation of C or, correspondingly, the amount of uncertainty about C; it is defined as

$$H(C) = -\sum_{c \in R_C} p(c) \cdot \log_2 p(c) = \sum_{c \in R_C} p(c) \cdot \log_2 \left(\frac{1}{p(c)} \right).$$

For a pair of random variables A and C, we measure the uncertainty about C after observing A by *conditional entropy*:

$$H(C|A) = \sum_{\substack{a \in R_A, \\ c \in R_C}} p(a,c) \cdot \log_2 \left(\frac{1}{p(c|a)} \right),$$

where $p(a,c) = P\{A = a, C = c\}$ and $p(c|a) = P\{C = c|A = a\}$. Now, since $H(C)$ represents our uncertainty about C before observing A, and $H(C|A)$ represents our uncertainty afterwards, their difference represents the amount of information about C given by A. This quantity is the *mutual information*, defined as

$$I(C,A) = H(C) - H(C|A).$$

In tree growing we, naturally, want to evaluate for each candidate attribute the given amount of information about the value of the class attribute. The attribute that increases information the most is then added to a node of the evolving tree. Generally we do not know the *prior probabilities* $p(c)$, $p(a,c)$, and $p(c|a)$, but have to resort to *data priors* $\hat{p}(c)$, $\hat{p}(a,c)$, and $\hat{p}(c|a)$ instead. Data priors are simply relative frequencies of attribute values in the data. For instance, $\hat{p}(c) = |S_c|/|S|$, where S is the (sub)sample under consideration and S_c is the subset of S, which consists of those elements that have $C = c$. (For a full account of data priors in this task see e.g., [Pagallo & Haussler 1990].) □

Another detail in decision tree growing, which deserves some attention, is that usually decision trees are generated without look-ahead and attributes are evaluated one at a time in separation, as presented above. In other words, if there is some dependence between two or more attributes (with respect to the classification of examples) this will go unnoticed by the evaluation function. It is only after one of the attributes is chosen (maybe by pure chance) to the tree that such interrelations between attributes may become visible to the evaluation function. A well-known example of functions with such dependent attributes is the family of multiplexor functions for which size-optimal decision trees are inherently unlearnable by algorithms utilizing impurity measures [Quinlan 1988a].

This shortcoming of impurity measures has received much attention in the context of decision tree learning only fairly recently, the obvious solution being to consider some combination of attributes at a time. In general, this technique is known as *constructive* *induction* [Michalski 1983], because new high-level features are constructed from the original primitive attributes, or as *dynamic bias* [Utgoff 1986], since the inductive bias changes dynamically. In decision tree learning constructive induction was first studied by Matheus and Rendell [1989] and Pagallo and Haussler [1990], both of whom used conjunctions of (binary) attributes as high-level features. A more traditional way to detect attribute interconnections is, as indicated above, to let the search procedure do *look-ahead*; i.e., instead of evaluating a single attribute at an isolated node of the evolving tree, evaluate multiple variables and (partial) subtrees at once. Naturally, this approach can turn out to be computationally quite expensive in comparison with the straightforward approach. Look-ahead is the approach taken, e.g., in the successful ACLS decision tree learner [Shepherd 1983, Shepherd *et al.* 1988], and it is, also, the approach taken in the development of a learning algorithm in the next chapter.

Similarly as attributes could be grouped together to form new features, using attribute value combinations can prove to be beneficial. To an extent it may prevent the same subtree from repeating at different branches of the tree and, more importantly, may retain the good functioning of the attribute selection heuristics—by "normalizing" the evaluation—when the size of the value range grows [Cestnik *et al.* 1987, Quinlan 1993]. On the downside, the number of candidate partitionings grows exponentially.

As a final point about the procedure *GrowTree* we note that it only specifies how to deal with nominal attributes; what to do with ordered and continuous, or *numerical*, value ranges is left open. Obviously, all values of an infinite range cannot have their individual edge and subtree in a decision tree, neither is it possible to let a finite but large number of values to be used. Therefore, numerical ranges need to be *categorized* into a small number of intervals. Categorization can be done by the domain expert in a knowledgeable way, or it can be left for the learning algorithm to do. Traditionally, in automatic techniques only a single *cut point* has been searched for and the value range has been divided into two in one node; this is known as the *binarization* technique [Breiman *et al.* 1984, Cestnik *et al.* 1987]. A range that is cut into two can, deeper down the tree, be refined further by binarization of the subranges. Recent studies [Chou 1991, Fayyad & Irani 1993, Fulton *et al.* 1995], however, indicate that it would be better to perform a multi-interval split at once rather than as a sequence of binary splits. Subsequently we assume that numerical ranges have been categorized beforehand. For more information on automatic handling of numerical ranges see, e.g., Fayyad and Irani [1992].

In the foregoing it has been presented only that, given a set of examples, a small decision tree that is consistent with the training set is to be sought for. This can hardly be termed learning. Anyhow, growing a consistent decision tree tends to produce hypotheses that are too specifically oriented to classifying those examples that happen to be present in the training set, in which case the tree grown will predict poorly the class of an unseen instance. In other words, the learning algorithm has not succeeded in capturing the underlying law governing the classification of instances and, thus, learning has

Procedure 2.4 *StoppingCondition*(S, C)
input: a set of examples S.
output: a truth value. (Out parameter C records a class name.)
begin
% Let C be the label of the majority of examples in S.
(1) **if** all but an *insignificant* portion of the examples in S belong to the class C **or**
(2) the cardinality of S is *too small* to calculate reliable significance measures
(3) **then return true else return false fi**
end;

not taken place. This phenomenon has been termed *overfitting* or *overgrowing*.

In order to avoid fitting the resulting tree too closely to the training set, it has to be *pruned* back to reduce the tree's dependence on the specialities of the training set. Alternatively, (preliminary) pruning can be performed simultaneously with the growing of a tree. Then, at each branch, growing has to be stopped before the tree becomes too specific. From the basic tree growing method we obtain procedure *GrowPrunedTree* by slightly modifying the stopping condition so that it leaves the tree robust with respect to the training set, but increases its accuracy in classifying new instances. Note that, while the basic tree growing procedure requires a consistent sample in order to be able to operate, the modified program tolerates inconsistencies in the training data. This is a necessary feature in noisy and incomplete domains of the real world.

The procedure *StoppingCondition* is changed to return **true** (leave a set of examples undivided) if the proportion of examples that have different class than the majority of examples or the absolute number of examples in the set is below a certain limit. Otherwise there is no change in the tree-growing procedure. A statistical distribution can be used to determine whether the proportion of examples, whose classification differs from that of the majority, is significant or not.

Example. Let A be an attribute with the (nominal) range $R_A = \{a_1, a_2, \ldots, a_m\}$ and let C be the class attribute with the range $R_C = \{c_1, c_2, \ldots, c_n\}$. The numbers of observed occurrences of A's values and classes can be cross-tabulated into a $m \times n$ contingency table as follows.

The deviation of the example distribution within the sample from the one expected, under the assumption that A is irrelevant to the class of an example, can be approximated from the data as the sum of the squared difference of the observed and expected number of occurrences of class $c_j \in R_C$ among the examples having value $a_i \in R_A$. The expected values are obtained by considering the whole subsample S. If A is irrelevant to the value of C, the expected number of occurrences with $A = a_i$ and $C = c_j$ in S is

Table 2.1: Cross-tabulation of classes and the values of attribute A.

	a_1	a_2	...	a_m	TOTAL
c_1	n_{11}	n_{21}	...	n_{m1}	N_1
c_2	n_{12}	n_{22}	...	n_{m2}	N_2
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
c_n	n_{1n}	n_{2n}	...	n_{mn}	N_n
TOTAL	M_1	M_2	$\ldots$	M_m	N

$e_{ij} = (M_i \times N_j)/N$, where $N = |S|$. The observed values, here, are the entries n_{ij} in the table above. Hence, the approximation of deviation is

$$\sum_{a \in R_A} \sum_{c \in R_C} \frac{(n_{ac} - e_{ac})^2}{e_{ac}} = \sum_{a=1}^{m} \sum_{c=1}^{n} \frac{(n_{ac} - e_{ac})^2}{e_{ac}}.$$

This statistic is distributed as the χ^2 distribution with $(m-1) \times (n-1)$ degrees of freedom. The χ^2 test for stochastic independence can now be used to determine, with desired confidence, whether attribute A is significant to the classification of examples or not. The difference of the statistic as computed from the data and as tabulated determines the confidence with which one can reject the hypothesis that the class of an example is independent of the value of A. If the confidence does not attain a user-specified threshold, attribute A is not accepted to the evolving tree. The χ^2 test, like most stochastic methods, looses its reliability when the number of examples becomes small [Quinlan 1991]. The above-described method is used in the ID3 algorithm of Quinlan [1986a] in significance testing and in a different task in the CN2 decision list learner [Clark & Niblett 1989]. □

Post-pruning of the constructed decision tree is performed by considering for each internal node of the tree whether it is better to leave the subtree rooted at that node intact or to replace it by a leaf node. This decision is based on an estimate of the change in error introduced by the replacement. Since the tree has been grown to faithfully reflect the composition of the training set, it cannot usually be pruned using error estimates arising from that same set of examples (*resubstitution errors*)—rather, an independent set of examples, a *pruning set*, is required. The error estimates utilized in pruning techniques are based on a variety of criteria. For instance, *cost-complexity pruning* [Breiman *et al.* 1984] takes into account the topology of the resulting tree in addition to the change in classification performance, *reduced error pruning* [Quinlan 1987c] (see the example below) only considers classification error, *pessimistic pruning* [Quinlan 1987c] uses binomial distribution with continuity correction, and *minimum error pruning* [Niblett & Bratko 1986] is based on Laplace's law of succession.

Example. *Reduced error pruning* is a method suggested by Quinlan [1987c] for decision tree simplification; it has recently found success as a pruning technique in rule set learning [Pagallo & Haussler 1990, Cohen 1993]. The method is straightforward: Given a decision tree T and a pruning set P, for every nonleaf subtree Q of T, examine the change in the number of misclassifications over P that would occur if Q was replaced by the best possible leaf. If the pruned tree would give equally many or fewer errors than T, then replace Q by the leaf. The process continues until no further replacements occur. □

The advantage post-pruning possesses over on-line pruning is that, when processing a fully grown decision tree, its global properties can be taken into account; when processing an evolving decision tree on the fly we must content ourselves with the local information available. Even combined pruning techniques exist: Gelfand *et al.* [1991] present a tree growing procedure that performs post-pruning on the fly by means of binarization: The training examples falling to the left subtree are used in pruning the right one, and *vice versa*. Clearly, harmful dependencies can invalidate this approach. In spite of post-pruning's clear advantage over on-line pruning we stick to the latter, because—as demonstrated in Chapter 3—it can be analytically shown to work. For further information and empirical comparisons of post-pruning techniques see, e.g., Mingers [1989b] or Buntine and Caruana [1993].

2.2 Alternative approaches to decision tree learning

The term "decision tree" is quite heavily overloaded in computer science; several fields have studied decision tree construction (see e.g., [Moret 1982]) and slightly different things are referred to as decision trees. There are, however, three (partially) separate communities that mean more or less the same thing with that term; the study of decision tree learning has progressed, in part, independently in these communities. Decision trees were first studied in statistical pattern recognition research. Machine learning borrowed the initial ideas from this community, but developed the ideas further independently. The latest recruit is the theory of computing community, which has come up with new viewpoints to decision tree learning quite recently.

The studies of pattern recognition and machine learning communities have led to quite similar results, though via different routes. The pattern recognition research is mainly concerned with numerical data and, since statistical discrimination is the classical technique for classifying numerical data, it is natural that trees, where the nodes contain a numerical discriminator, *regression trees* [Breiman *et al.* 1984], for instance, were the starting point of decision tree studies in this field (see e.g., [Henrichon & Fu 1969, Friedman 1977]). On the other hand, machine learning—as a subfield of artificial intelligence—is more interested in symbolic data and, therefore, a different approach was first taken [Quinlan 1983]. Later these fields have come closer together; methods for handling symbolic data have been incorporated into pattern recognition techniques [Breiman *et al*. 1984] and numerical databases have gained importance in machine learning [Van de Merckt 1993, Elomaa & Ukkonen 1994, Murthy *et al*. 1994].

The research on decision tree learning is totally dominated by the recursive partitioning control strategy of TDIDT in both pattern recognition and empirical machine learning. Rather than alternative approaches, these fields have studied learning alternative, but closely related, concept representations. For example, learning algorithms for *production rule sets* [Michalski *et al*. 1986] and *decision lists* [Clark & Niblett 1989] use pretty much the same induction techniques as TDIDT and these concept representations have a direct correspondence to decision trees. Learning true extensions of decision trees has also been studied: *Decision trellises* [Chou 1991] and *decision graphs* [Oliver 1993, Kohavi 1994] have the form of a directed acyclic graph (dag)—a tree is a special case of a dag. Quinlan [1990b] has even applied basic TDIDT techniques to *inductive logic programming*.

Bottom-up control strategy for decision tree construction has been suggested by Landeweerd *et al*. [1983], but since their approach is unsupervised, it has only little relevance to other decision tree learning approaches. Another concept learning approach that could be seen as an alternative to TDIDT is constructing *one-level*, or *one-shot*, decision trees [Landeweerd *et al*. 1983, Iba & Langley 1992, Holte 1993] and other similar subclasses defined by extreme syntactic restrictions [Auer *et al*. 1995]. However, basically they are just restricted forms of TDIDT and as such do not qualify as true alternatives. Furthermore, the utility and potential of such approaches have serious limitations [Elomaa 1994].

Rissanen's [1989] *minimum description length principle* (MDLP) says that the best explanation of a set of data is the one that minimizes the representation length (in bits) of the data when represented in terms of a theory and exceptions to it (cf. Occam's Razor). This approach has recently gained much ground in machine learning. The first serious attempt to apply MDLP to decision tree learning was done by Quinlan and Rivest [1989]; Wallace and Patrick [1993] later continued the work. In this connection the theory, of course, has the form of a decision tree. Hence, efficient encoding of a decision tree is essential in this approach, and that is what Quinlan and Rivest as well as Wallace and Patrick consider most. The search control strategy in their decision tree learning method, though, is the familiar two-stage TDIDT approach, where MDLP determines the attribute evaluation function. Rissanen's [1995] own studies on MDLP-based decision tree learning follow closely those of Quinlan and Rivest. Fayyad and Irani [1993] and Quinlan [1996] have also applied MDLP to decision tree learning. They, too, use standard TDIDT control strategy and put MDLP into practice only when choosing the cut points in a continuous-valued attribute's discrete categorization.

Also in pattern recognition many methods for learning decision trees that fulfill some optimality criterion have been studied. For example, Meisel and Michalopoulos [1973] have studied learning decision trees that minimize the average path length. Payne and Meisel [1977] have studied generic optimal decision tree construction, where a partitioning of the data and an optimality criterion are given and a decision tree, which is equivalent with the given partitioning, but minimal with respect to the criterion, is developed. Because of the known computational intractability of learning optimal decision trees, these methods, however, fail to fulfill practical efficiency requirements.

Theoretical work on machine learning has, naturally, had an interest in providing results about the learnability of the concept representations that are applied in practice (the formal learning model is introduced in the next section). In the mathematical learning framework the hypothesis does not necessarily have the same form as the target concept—it may belong to another (a richer) class of representations. Therefore, many theoretical studies are not quite along the same line as empirical ones, where the hypothesis representation language—suiting the purpose better or worse—is usually fixed. Nevertheless, the general learnability of decision trees remains an open problem even today [Hancock *et al*. 1995]. However, in the tradition of empirical research Ehrenfeucht and Haussler [1989] provided a constructive proof of the learnability of a subclass of decision trees. We consider this subclass and present extensions to the learning method in the next chapter. A closely related result is Rivest's [1987] proof of the learnability of k-decision lists—a subset of general decision lists.

Linial *et al*. [1989] initially showed how learnability under the uniform example distribution (cf. next section) can be derived by observing the spectrums of the Fourier transforms of a function class. The key idea in this technique is that learning can be achieved by estimating the dominating Fourier coefficients from randomly chosen inputs. The first polynomial-time learning results using this technique were provided for decision trees. First, Aiello and Mihail [1991] proved that μ-decision trees are polynomial-time learnable; a μ-decision tree mentions any variable at most once. Soon thereafter Kushilevitz and Mansour [1991], using the same technique, obtained the learnability of decision trees that may have linear operations (features) in each node; the learning algorithm is permitted—and required—to ask membership queries (cf. [Angluin 1988]). Both proofs are constructive, i.e., a learning algorithm is presented, but the hypotheses do not have the form of a decision tree. Furthermore, both results only apply when all examples of the domain are equally likely, i.e., under the uniform distribution.

Hancock [1990, 1991, 1993] has studied the learnability of decision trees where the number of variable occurrences is limited—in the simplest form μ-decision trees and, in general, $k\mu$ decision trees, where k denotes the variable occurrence restriction. In these studies membership and equivalence queries are allowed—and needed. This work has demonstrated, for an arbitrary constant k, the learnability of $k\mu$ decision trees over arbitrary example distributions using richer hypothesis description languages [Hancock 1991] and over the uniform example distribution using only decision trees as hypotheses

[Hancock 1993].

2.3 Probably approximately correct learning

Valiant's [1984] attempt to formalize inductive concept learning has drawn attention to the computational aspects of machine learning. In particular, Valiant's framework emphasizes the efficiency of hypothesis building. Learning in the sense of Valiant's definition is known as *probably approximately correct* (PAC), or *distribution-free*, learning. Despite the evident shortcomings of the PAC learning framework's underpinnings and utility [Amsterdam 1988a, 1988b, Buntine 1989, 1990, Dietterich 1989, Saitta & Bergadano 1993] it has become the standard theoretical model of concept learning.

Several different formulations of PAC learning have appeared in the literature. The variants, however, turn out to define equal models [Haussler *et al.* 1991]. In the following we recapitulate Natarajan's [1991] formulation of PAC learning of functions on discrete domains. For an extensive treatment of computational learning theory we refer the reader to one of the recent textbooks on the topic; e.g., to that by Anthony and Biggs [1992] or that by Kearns and Vazirani [1994]. For a more concise presentation of the research issues see Angluin's [1992] article.

In the following we consider multivalued variables and classes. It is assumed that a finite discrete domain of multiple values is encoded with a sequence of consecutive natural numbers starting from 1. Such an initial segment is denoted $[m] = \{\, i \in \mathbb{N}_+ \mid i \leq m \,\}$. For example, the domain $\mathbf{D} = \{a, b, c, d, e\}$, whose cardinality is 5, is encoded as $[5] = \{1, 2, 3, 4, 5\}$. We are dealing with nominal attributes only; i.e., the values are unordered. By $[m]^n$ we denote, as usual, the n-fold Cartesian product of $[m]$. A n-ary function on $[m]$ is a mapping from $[m]^n$ to $[m]$.

Notation We denote the symmetric difference of two sets S and T by $S \triangle T = (S \setminus T) \cup (T \setminus S)$. The cardinality of a set S is denoted by $|S|$. The set of all finite strings of an alphabet Σ is denoted by Σ^*. If $w \in \Sigma^*$, then the length of w, denoted by $|w|$, is the number of characters in the string w. We let $\Sigma^{[n]}$ denote the set $\{\, w \in \Sigma^* \mid |w| \leq n \,\}$. By ${}^{B}\!A$, where A and B are sets, we denote the set of functions $\{\, f \mid f \text{ is a function, } \mathrm{Dom}(f) = B, \mathrm{Rng}(f) \subseteq A \,\}$.

A *concept* γ is a subset of the *instance space* X, where X is an arbitrary set. Let χ_γ be the *indicator function* associated with γ; this function indicates for each $x \in X$ whether it belongs to γ or not:

$$\chi_\gamma(x) = \begin{cases} 1 & \text{if } x \in \gamma, \\ 0 & \text{if } x \notin \gamma. \end{cases}$$

An *example* of a total function $\phi\colon X \to Y$ is a pair $\langle x, \phi(x) \rangle$, where $x \in X$. The set of all examples for ϕ is graph(ϕ) $= \{ \langle x, y \rangle \mid x \in X,\ y = \phi(x) \}$. We say that ϕ is *consistent* with a set of examples S if $S \subseteq \mathrm{graph}(\phi)$.

A *function class* is a triple $\mathcal{F} = (F, X, Y)$, where X and Y are arbitrary sets and F is any collection of total functions from X to Y. The set X is the *domain* of $\mathcal{F}$ and Y is the *range* of $\mathcal{F}$. They will also be called the *instance space* and the *set of classes*, respectively. A class of representations for functions is a five-tuple: $\mathbf{R} = (\Sigma, \Pi, \Gamma, R, f)$. Sets Σ, Π, and Γ are finite alphabets. Strings composed of characters in Σ are used to describe elements of X, strings in Π^* are used to describe elements of Y, and strings in Γ^* describe the functions. The set $R \subseteq \Gamma^*$ is the collection of function representations, and $f\colon R \to^{\Sigma^*} \Pi^*$ is a mapping from these representations into functions from Σ^* to Π^*. For any representation $r \in R$ by $f(r)$ we denote the function represented by r. For any class of representations $\mathbf{R} = (\Sigma, \Pi, \Gamma, R, f)$ there is an associated function class $\mathbf{F}(\mathbf{R}) = (f(R), \Sigma^*, \Pi^*)$, where $f(R) = \{ f(r) \mid r \in R \}$. The length of the shortest representation for a function ϕ is denoted by $\ell_{\min}(\phi, \mathbf{R}) = \min \{ |r| \mid r \in R,\ f(r) = \phi \}$ (or $\ell_{\min}(\phi)$ for short when $\mathbf{R}$ is clear from the context).

Example. Recall our mail order enterprise from the examples of Chapter 1. The customer profile descriptions represented by decision trees are total functions. Hence, the *function class* ($\mathcal{DT}$, "customer","product group") is what our company is interested in. The total functions in $\mathcal{DT}$ are defined by decision trees fulfilling some criterion; for example, our enterprise may only be interested in errorless descriptions, i.e., consistent trees. Let us assume, for simplicity, that customers are described by n attributes each having m possible values and that there are k different product groups that we are interested in. Then, a customer is described by giving, for each attribute, the number in $[m]$ that corresponds to the observed value of the attribute. Thus, instances are strings in $\{1, \ldots, m\}^n$ and, hence, $\Sigma = \{1, \ldots, m\}$. Let Υ be an alphabet for writing attribute names down; e.g., $\Upsilon = \{$ 'sex', 'population', 'distance', ... $\}$. By tabulating the attribute names it suffices to set Υ to be the set of table indices, i.e., $\Upsilon = [n]$. Similarly, Π is an alphabet for describing the set of class labels; here it suffices to have $\Pi = [k]$. Γ is the set of characters needed to represent trees, e.g., in the *nested parentheses representation* [Knuth 1969]: $\Gamma = \Upsilon \cup \Pi \cup \{$ '(', ')' $\}$. Then the set R consists of legal nested parentheses representations of decision trees over $\Upsilon \cup \Pi$, where strings of characters of Υ are used to label internal nodes and characters of Π are used as leaf labels. Finally, function f maps each decision tree representation $r \in R$ to the unique mapping, which is a total function from Σ^n to Π such that it satisfies all paths of the decision tree corresponding to r. We say that a function g satisfies a path in a decision tree if g maps a configuration corresponding to the edge labels on the path to the label of the leaf node at the end of the path. □

If $f(r)\colon x \mapsto y$, then we write $r(x) = y$. We write r in place of $f(r)$ when the meaning is clear from the context. An *example* of r is a pair $\langle x, r(x) \rangle$. Here the learning algorithm has at its disposal an *example oracle* $EX(P, r)$, which draws examples of the target function $f(r)$ according to the fixed, but unknown probability distribution P on Σ^*. In other words, when called, the oracle chooses a description of an instance $x \in \Sigma^*$, according to P, and returns the pair $\langle x, r(x) \rangle$.

The *error* of a hypothesis function h is the total weight of the unknown, arbitrary probability distribution P on the instances that are mapped incorrectly by h. In other words, for the target function t the error of hypothesis h is

$$P(h \bigtriangleup t) = \sum_{\Theta} P(x),$$

where $\Theta = \{ x \in X \mid h(x) \neq t(x) \}$. Note that misclassifying rare cases causes less error than misclassifying commonly occurring cases.

In the formal definition of learning we need to be exact about the lengths of input and output strings. Therefore, we need to refer to length-bounded subsets of function classes. In the following we define projections of functions and function classes.

Definition For any function $g \in f(R) \in \mathbf{F}(\mathbf{R})$, for some $\mathbf{R}$, and for any $n, k \in \mathbb{N}$, the *projection* $g_{n,k}$ of g on $\Sigma^{[n]} \times \Pi^{[k]}$ is

1. undefined if there exists $x \in \Sigma^{[n]}$ such that $g(x) \notin \Pi^{[k]}$;
2. else, the function $g_{n,k}: \Sigma^{[n]} \to \Pi^{[k]}$ such that for all $x \in \Sigma^{[n]}$, $g_{n,k}(x) = g(x)$.

The subclass $F_{n,k}(\mathbf{R})$ of $\mathbf{F}(\mathbf{R})$ is the projection of $\mathbf{F}(\mathbf{R})$ on $\Sigma^{[n]} \times \Pi^{[k]}$; i.e., $F_{n,k}(\mathbf{R}) = (f_{n,k}(R), \Sigma^{[n]}, \Pi^{[k]})$, where

$$f_{n,k}(R) = \{ g_{n,k} \mid g_{n,k} \text{ is defined, } g \in f(R) \}.$$

By $f_{n,k}(r)$ we denote the projection of $f(r)$ on $\Sigma^{[n]} \times \Pi^{[k]}$.

Let us make the following simple observation. Let h be a hypothesis function that is consistent with a sample of the target function $t \in F_{n,k}(\mathbf{R})$, for some $n, k \in \mathbb{N}$, drawn according to a probability distribution P on $\Sigma^{[n]}$. Now, P is nonzero only on strings of length n or less. All the examples drawn involve strings from $\Sigma^{[n]}$, and h is consistent with these examples if and only if $h_{n,k}$ is consistent with them. Therefore $P(h \bigtriangleup t) = P(h_{n,k} \bigtriangleup t_{n,k})$.

At last we have at our disposal all the necessary notation and concepts that are needed in formalizing probably approximately correct learning. There are two parts to PAC-learning: The target concept has to be identified accurately with a high probability and it has to happen efficiently (in polynomial time). If there exists a learning algorithm that can attain both requirements for all projections $F_{n,k}(\mathbf{R})$ of a function class **F**(**R**) under any probability distribution on the instance space, then we say that **R** is polynomially learnable.

The success of a learning algorithm is measured by two parameters which are supplied as inputs to the algorithm. The algorithm is expected to produce a hypothesis h, whose error is less than an arbitrary prespecified *accuracy parameter* ε, where $0 < \varepsilon \leq 1$. The rate on which the algorithm is to produce these accurate hypotheses is specified with a *confidence parameter* δ, where $0 < \delta \leq 1$. Because of "bad luck" in drawing the examples the algorithm may sometimes produce an answer with error greater than ε, but we require that the probability that an accurate answer is produced is at least $1 - \delta$. The tighter the bounds ε and δ are, the more examples and computation time the algorithm is expected to consume.

The time complexity of the learning algorithm is further restricted by the length of the input examples and the length of the shortest name of the target function f: $\ell_{\min}(f)$. Polynomial learnability of function class representations is defined as follows.

Definition 1 A class of representations $\mathbf{R} = (\Sigma, \Pi, \Gamma, R, f)$ is *polynomially learnable*, if there exists an algorithm L and a polynomial p_L such that for all

- $n, k \geq 1$,
- ε and δ, where $0 < \varepsilon, \delta \leq 1$,
- $r \in R$ such that $f(r) \in F_{n,k}(\mathbf{R})$, and
- probability distributions P on $\Sigma^{[n]}$,

if L is given as input the parameters n, k, ε, and δ, and may access the oracle $EX(P, r)$, then L halts in time $p_L(n, k, \ell_{\min}(f(r)), 1/\varepsilon, 1/\delta)$ and, with probability at least $1 - \delta$, outputs a representation $r' \in R$ such that $P(r' \triangle r) \leq \varepsilon$. Such an algorithm L is a *polynomial-time learning algorithm* for $\mathbf{R}$.

After having made the representation issues formal, we now turn to functions from Σ^* to Π^* and the learnability of classes of such functions. The underlying idea is that there is a fixed class of representations $\mathbf{R}$ that we are concerned with. In the following we write $F_{n,k}$ instead of $F_{n,k}(\mathbf{R})$ and $\mathbf{F}$ instead of $\mathbf{F}(\mathbf{R})$. We note that the following consequence of the above definition of polynomial learnability carries from concept learning over to function class learning. The result was originally presented by Blumer *et al.* [1987].

Theorem 2 *Let F be a function class, and let P be a probability distribution. Given a function $g \in F$ and a sample S of g of size m, drawn according to the probability distribution P, the probability is at most*

$$|F|\,(1 - \varepsilon)^m$$

that there exists a function $h \in F$ such that the error of h is greater than ε, and h is consistent with S.

Proof If h is a single function in F of error greater than ε for the target g, the chance that h is consistent with a random sample S of size m is less than $(1-\varepsilon)^m$. Since F has $|F|$ members, the chance that there exists *any* member of F that is consistent with the sample and satisfies the error condition is at most $|F|(1-\varepsilon)^m$. □

For a function class F to be polynomially learnable, the probability of coming across a function with error greater ε may be at most the value of the confidence parameter δ. Hence, from

$$|F|(1-\varepsilon)^m \leq \delta$$

we obtain that

$$m \geq \frac{1}{-\ln(1-\varepsilon)}\left(\ln|F| + \ln\frac{1}{\delta}\right),$$

which certainly holds if

$$m > \frac{1}{\varepsilon}\left(\ln|F| + \ln\frac{1}{\delta}\right),$$

since $\ln(1+\alpha) < \alpha$. If m satisfies the above inequality, then the probability is at most δ that a function in F, which is consistent with S, will turn out to have error greater than ε. Hence, as a corollary we obtain the following result.

Corollary 3 *Let F be a function class, and let P be a probability distribution. Consider any ε and δ such that $0 < \varepsilon, \delta \leq 1$, and any target function g in F, and a sequence of at least*

$$\frac{1}{\varepsilon}\ln\frac{|F|}{\delta}$$

random examples of g, each chosen independently according to P. Then with probability at least $1-\delta$, every function $h \in F$ that is consistent with all of these examples has error at most ε for the target g. □

By this result, if there exists an algorithm, which can identify a hypothesis that is consistent with a sample of $(1/\varepsilon)\ln(|F|/\delta)$ random examples, then that algorithm is a polynomial learning algorithm for F if only the identification happens in polynomial time.

This formulation brings out a quite natural measure of the complexity of a function class: The logarithm of the size of the function class, $\ln|F|$. It may be viewed as the number of bits needed to write down an arbitrary element of F using an optimal encoding [Blumer *et al.* 1987, Rivest 1987]. Abusing terminology slightly we say that a function class $F_{n,k}$ is *polynomial-sized* if the complexity measure $\ln|F_{n,k}|$ is bounded by a polynomial in n and k. Natarajan [1991] uses a generalization of the Vapnik-Chervonenkis (VC) dimension [Blumer *et al.* 1987, 1989] to express his results. Instead of using this somewhat abstract measure, we use the more concrete complexity measure $\ln|F|$ to express the numbers of required examples. This gives often better approximation results than using the VC dimension.

2.4 Coping with random classification noise

Several derivates of the basic PAC-framework for modeling noise-affected learning situations have been put forward since Valiant's [1984] original learnability definition (e.g., [Angluin & Laird 1988, Boucheron & Sallantin 1988, Kearns & Li 1988, Laird 1988, Shackelford & Volper 1988, Sloan 1988, Valiant 1985]). In this section we show how one of these models can be extended to deal with function classes.

In Angluin and Laird's [1988] random classification noise model the learning algorithm requests examples of the target concept f from a sampling oracle $EX_\eta(P, f)$, where P is the unknown probability distribution on the instance space and $\eta < 1/2$ is the unknown *noise rate* affecting the oracle. The oracle $EX_\eta(P, f)$, when called, draws an instance x according to P and, with probability $1 - \eta$, returns the pair $\langle x, f(x)\rangle$ and, with probability η, the pair $\langle x, 1 - f(x)\rangle$. Learnability in the presence of random classification noise is defined equivalently to polynomial-time learnability [Valiant 1984], except that now an additional parameter—an upper bound η_b for the noise rate—has to be taken into account in a learning algorithm's time complexity. The noise rate is taken notice of by bounding the value of $(1/2 - \eta_b)^{-1}$ by a polynomial.

To extend the classification noise model into multiconcept learning situations we let the oracle $EX_\eta(P, f)$ return, with probability η, an erroneous classification for the selected instance. For our considerations it is immaterial how the erroneous value is determined; for the sake of completeness, let us, however, agree that all $k - 1$ incorrect classes have the same probability $\eta/(k - 1)$ of being chosen. The noise rate η must be restricted below value $1/2$ as in Angluin and Laird's original model in order for an identification procedure to be able to work (cf. [Angluin & Laird 1988, p. 348]).

Note that we are still effectively dealing with binary functions here: Either the example returned by the oracle has the correct label or it has a label that is incorrect, be it of any class. Thus, most results on noise-tolerant learning—in particular, those by Sakakibara [1991, 1993]—apply in this situation too. Furthermore, even the proofs are essentially the same and, hence, will be omitted here (see Appendix A for relevant proofs).

The meaningfulness of this noise model and its relation to real life may well be questioned but, nevertheless, this type of corruption of values has widely been used to test practical learning algorithms (e.g., [Clark & Niblett 1989, Quinlan 1986a, 1986b]) and it provides a proper extension of Angluin and Laird's model. Here is the definition of learnability in the presence of random classification noise.

Definition 4 The class of representations $\mathbf{R} = (\Sigma, \Pi, \Gamma, R, f)$ is *polynomially learnable in the presence of classification noise* if there exists an algorithm L and a polynomial p_L such that for all

- $n, k \geq 1$ and $\eta < 1/2$,

- ε and δ, where $0 < \varepsilon, \delta \leq 1$,
- $r \in R$ such that $f(r) \in F_{n,k}(\mathbf{R})$, and
- probability distributions P on $\Sigma^{[n]}$,

if L is given as input the parameters n, k, ε, δ, and η_b, such that $\eta \leq \eta_b < 1/2$, and may access the oracle $EX_\eta(P, r)$, then L halts in time

$$p_L\left(n, k, \ell_{\min}\left(f(r)\right), \frac{1}{\varepsilon}, \frac{1}{\delta}, \frac{1}{1/2 - \eta_b}\right)$$

and, with probability at least $1-\delta$, outputs a representation $r' \in R$ such that $P(r' \triangle r) \leq \varepsilon$.

In noise-affected domains examples drawn from the oracle all have to be taken into account. We cannot treat the sample as a *set* of examples. Instead, the multiple occurrences of examples in the sequence of examples drawn from the oracle have to be dealt with. However, the order of examples is not important here and, thus, we can treat the sample as a multiset. We let $|S|$ denote the total number of examples (including multiple occurrences) in the sample S. In addition, we let $D(f, S)$ denote the number of disagreements between the function f and the sample S, i.e., if $S = \langle x_1, l_1\rangle, \langle x_2, l_2\rangle, \ldots, \langle x_q, l_q\rangle$, then $D(f, S)$ is the number of indices j for which $f(x_j) \neq l_j$.

Starting directly from the definition of learnability in the presence of classification noise, it can be very difficult to obtain positive results (cf. [Angluin & Laird 1988]). A helpful vehicle in proving the learnability of decision trees in the presence of classification noise are *noise-tolerant Occam algorithms*—Sakakibara's [1993] generalization of Occam algorithms [Blumer *et al.* 1987, 1989, Board & Pitt 1992]. Informally, from a noise-tolerant Occam algorithm we require that the average disagreement in classifying an instance using its hypothesis is, with high probability, at most only slightly above the upper bound η_b for the noise rate. The average disagreement may climb over the noise rate only by an additive factor that is relative to the allowed error and the noise rate. A straightforward generalization of these algorithms to function class learning situations follows.

Definition 5 A *noise-tolerant Occam algorithm* O for a function class $\mathbf{F}$ is an algorithm that, when given as input a sufficiently large sample S of q examples drawn from $EX_\eta(P, f)$, where $f \in F_{n,k}$, and parameters ε, δ, and η_b,

1. produces a representation r of a function $h \in \mathbf{F}$, such that

$$\frac{D(h, S)}{q} \leq \eta_b + \frac{\varepsilon(1 - 2\eta_b)}{4},$$

with probability at least $1 - \delta/2$, and

2. runs in time that is polynomial in n, k, q, $1/\varepsilon$, $1/\delta$, and $1/(1-2\eta_b)$.

This definition is identical to Sakakibara's [1993] original formulation of noise-tolerant Occam algorithms, except that now we are dealing with function classes and the space $[m]^n$ rather than concept classes and the Boolean space.

Now the main result of Sakakibara [1993], namely that the existence of a noise-tolerant Occam algorithm implies learnability in the presence of classification noise, can be proved in the extended model also. The sample size that is required to guarantee, with high probability, a low error for the resulting classifier is only four times that which is required by an algorithm that works by minimizing disagreement [Angluin & Laird 1988].

Theorem 6 *Let* **F** *be a polynomial-sized function class and let* η_b *be such that* $\eta \leq \eta_b \leq \eta + \varepsilon(1-2\eta)/2$. *If there exists a noise-tolerant Occam algorithm for* **F**, *then* **F** *is polynomially learnable in the presence of classification noise. The sample size required is at least*

$$\frac{8}{\varepsilon^2(1/2-\eta_b)^2}\ln\frac{2|F_{n,k}|}{\delta}.$$

Proof Simple modification of Sakakibara's [1993] proof (see appendix A for full proof). The key idea in the proof is that if the noise-tolerant Occam algorithm is provided with a close-enough approximation of the noise rate η and a large-enough sample, then the hypothesis returned by the algorithm fulfills the conditions of Definition 4. The sample can be queried from the oracle and a good approximation for η can be found by iterating the algorithm with carefully chosen, successively smaller values of η_b. □

Chapter 3

The Design of a Learning Algorithm

Ehrenfeucht and Haussler [1989] have shown that a subset of decision trees can be learned in the PAC learning framework. The concept representations within this subset are decision trees whose *rank* is bounded. Ehrenfeucht and Haussler also exhibit a learning algorithm for these decision trees in the binary, noise-free setting. In this chapter we demonstrate how their algorithm can be developed into a practical learning tool without losing its provable properties. First, in Sections 3.1 and 3.2, we generalize the concept of rank and modify the algorithm to deal with multivalued variables and classes. Then, in Section 3.3, we demonstrate how decision trees of minimum rank can be constructed efficiently in the incremental setting. In Section 3.4 we modify the algorithm to cope with random classification errors in the training examples. Finally, we present a new decision tree learning algorithm—called *Rank*—incorporating all these properties.

3.1 The rank of a decision tree

We formalize first decision trees, their rank, and the functions they represent. Then a key lemma, on which the proof of the learnability of decision trees is essentially based on, is presented.

Definition Let $V_{n,m} = \{v_1, \ldots, v_n\}$ be a set of n m-ary variables. The class $\mathrm{DT}_m(n)$ of *m-ary decision trees* (over $V_{n,m}$) is defined recursively as follows:

1. If T is the m-ary tree consisting of a single node labeled with $k \in [m]$ then $T \in \mathrm{DT}_m(n)$. We denote this case $T = k$.

2. If $T_1, T_2, \ldots, T_m \in \mathrm{DT}_m(n)$ and $v \in V_{n,m}$, then the m-ary tree with root labeled v and with i-th subtree T_i, for all $i \in [m]$, is in $\mathrm{DT}_m(n)$. We refer to the i-th subtree as the *i-subtree*.

We say that a decision tree is *reduced* if each variable name appears at most once on any path from the root to a leaf.

A decision tree $T \in \mathrm{DT}_m(n)$ defines a total function $f_T\colon [m]^n \to [m]$ in a natural manner. Routing an example through a tree maps a set of variables in $V_{n,m}$ (the ones tested at the nodes en route from the root to a leaf) with certain values (the turns taken on the path) to a value for the function represented by the tree (the label of the leaf reached). More formally, we define f_T as follows:

1. If $T = k$, then f_T is the constant function **k**.
2. Else if v_i is the label of the root of T and T_j the j-subtree for all $j \in [m]$, then for any point $x = (x_1, \ldots, x_n) \in [m]^n$ we have: If $x_i = k$ ($k \in [m]$), then $f_T(x) = f_{T_k}(x)$.

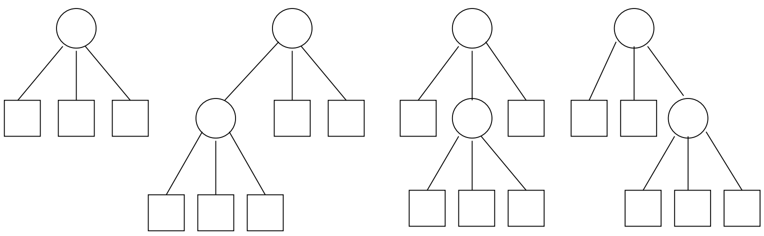
The following definition is the feasible one out of the natural ways of generalizing Ehrenfeucht and Haussler's [1989] definition of rank for binary decision trees (cf. [Elomaa 1992]). It states that a tree of rank r can have at most one subtree with rank r. The other subtrees may have rank at most $r - 1$.

Definition The *rank* of a reduced decision tree T, denoted $r(T)$, is defined as follows:

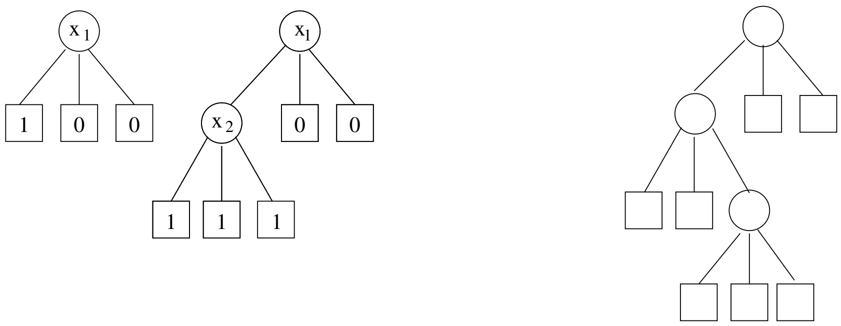
1. If T consists of a single leaf, then $r(T) = 0$.
2. Else if T_{max} is a subtree of T with the maximum rank r_{max}, then
$$r(T) = \begin{cases} r_{max} & \text{if } T_{max} \text{ is unique,} \\ r_{max} + 1 & \text{otherwise.} \end{cases}$$

We let $\mathrm{DT}_m^r(n)$ denote the set of all m-ary decision trees in $\mathrm{DT}_m(n)$ of rank at most r and we let $F_m(n, r)$ denote the set of n-ary functions on $[m]$ that are represented by the trees in $\mathrm{DT}_m^r(n)$.

Example. Delimiting the rank of a decision tree to value r (together with the value m) determines the tree structures in $\mathrm{DT}_m^r(n)$: $\mathrm{DT}_m^0(n)$ only contains the single-leaf tree structure, independent of the value of m. The number of possible labelings of that only leaf, n, then determines the number of functionally different equal-structured decision trees. In particular, $\mathrm{DT}_m^0(n)$ always contains n separate one-leaf decision trees. For values $r > 0$ there exists more than just one possible tree structure (assuming $n > 1$). For instance, the following picture illustrates all (reduced) tree structures contained in $\mathrm{DT}_3^1(2)$.

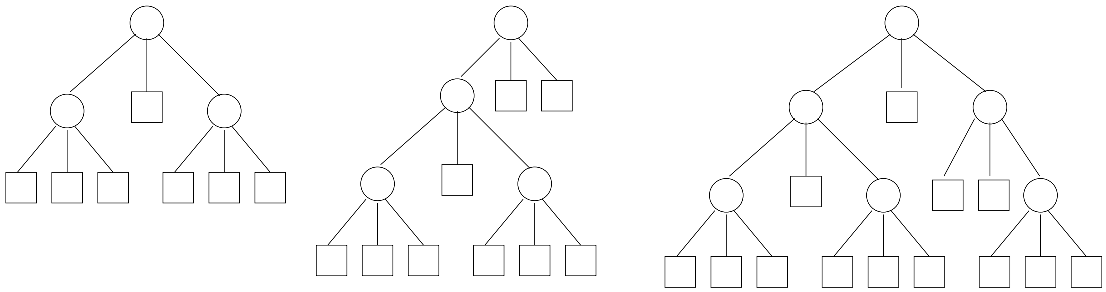


The actual decision trees are obtained from these structures by assigning each node with a label; the labeling must be legal, i.e., it has to keep the tree reduced. The function represented by a decision tree is not necessarily unique. For example, the two left-most decision trees in the following picture represent the same function.



If we increase the number of available variables, the one nontrivial subtree may grow further in size but not in rank. For instance, the right-most tree structure in the previous picture belongs to $\mathrm{DT}_3^1(3)$.

For higher values of rank the allowed tree structures are obtained by the same recursive definition: At most one subtree may have the same rank as the whole tree, but its position may vary. As the parameters m, n, and r grow, the number of legal tree structures and their possible labelings goes up quickly. The following tree structures are examples of those belonging to $\mathrm{DT}_3^2(3)$.



Clearly, for any reduced decision tree T of arity m on n variables there exists $r \geq 0$ such that $T \in \mathrm{DT}_m^r(n)$. In other words, $\bigcup_{r\geq 0} \mathrm{DT}_m^r(n)$ contains all legal reduced decision trees, but for any r there exists trees not enclosed in $\mathrm{DT}_m^r(n)$. $\square$

It is easily verified that $\bigcup_{r\geq 0} F_m(n,r)$ is the set of all functions from $[m]^n$ to $[m]$. However, for every fixed r, the set $F_m(n,r)$ is a proper subset of them. The following counting argument demonstrates this by giving an upper bound for the number of functions $F_m(n,r)$ when $r < n$.

Lemma 7

(i) Let k be the number of nodes in a reduced m-ary decision tree over $V_{n,m}$ of rank r, where $n \geq r \geq 1$ and $m \geq 2$. Then

$$2^r m - m + 1 \leq k \leq (m-1)^{-1}\left(m\sum_{i=0}^{r}\left(\binom{n}{i}(m-1)^i\right)-1\right) < 2\left(\frac{emn}{r}\right)^r,$$

where e is the base of the natural logarithm.

(ii) If r = 0 then $|F_m(n,r)| = m$, else

$$|F_m(n,r)| \leq \begin{cases} m^{m^n} & \text{if } n \leq r, \\ (4^m mn)^{(emn/r)^r} & \text{if } n > r. \end{cases}$$

Proof

(i) Since $n \geq r$, the number of variables allows constructing all trees of height $r+1$, which include the smallest decision tree of rank r. Thus, in this case, the size of the tree depends only on the values of r and m, not on n. Let $N(r,m)$ denote the number of nodes in the smallest m-ary decision tree of rank r. From the definition of rank, we have

$$\begin{aligned} N(1,m) &= m+1 \text{ for all } m \geq 2, \text{ and} \\ N(r,m) &= 2N(r-1,m)+m-2+1 \\ &= 2N(r-1,m)+m-1 \text{ for all } m \geq 2 \text{ and } r > 1. \end{aligned}$$

The solution to this recurrence is

$$N(r,m) = 2^{r-1}(m+1) + \sum_{i=0}^{r-2} 2^i(m-1) = 2^r m - m + 1.$$

Hence we have the first inequality.

Now let $L(n,r)$ denote the maximum number of leaves in any reduced m-ary decision tree over $V_{n,m}$ of rank r. Observing that the largest t-ary decision tree over n variables of rank n is the complete t-ary tree of height $n+1$ and that such a tree has t^n leaves, we clearly have, from the definition of rank, the recurrence system

$$\begin{aligned} &L(n,0) = 1 \text{ for all } n \geq 0, \\ &L(n,n) = m^n \text{ for all } n \geq 1, \text{ and} \\ &L(n,r) = L(n-1,r) + (m-1)L(n-1,r-1) \text{ for all } n > r \geq 1, \end{aligned}$$

since the variable that appears in the root of a reduced tree does not appear in any subtree of the root. It is verified that the solution of this recurrence for $n \geq r$ is

$$L(n,r) = \sum_{i=0}^{r} \left(\binom{n}{i} (m-1)^i \right),$$

which has the strict upper bound $(en/r)^r(m-1)^r < (emn/r)^r$ for all $n \geq r \geq 1$ (see [Blumer *et al.* 1989]). A full m-ary tree has exactly $(L-1)/(m-1)$ internal nodes, where L is the number of leaves in the tree. Hence, the total number of nodes is $L + (L-1)/(m-1) = (mL-1)/(m-1)$. This yields the second inequality. Noting that $m \geq 2$, we have that the number of nodes is bounded above by $2L < 2(emn/r)^r$ yielding thus the third inequality.

(ii) If $r = 0$ then $F_m(n,r)$ includes only the constant functions **1**, **2**, ..., **m**. Hence $|F_m(n,r)| = m$ in this case. If $n \leq r$ then $\mathrm{DT}_m^r(n)$ includes every full m-ary decision tree of depth n. Hence $F_m(n,r)$ includes all functions $f\colon [m]^n \to [m]$, and thus $|F_m(n,r)| = m^{m^n}$. If $n > r \geq 1$ then each function in $F_m(n,r)$ is represented by an m-ary tree with at most $k = (emn/r)^r$ leaves, as shown above. Let $p = m/(m-1)$, and note that $p \leq 2$ when $m \geq 2$. The number of distinct unlabeled t-ary trees with z nodes is [Knuth 1969, Exercise 2.3.4.4-11]

$$\frac{1}{(t-1)z+1}\binom{tz}{z} = \frac{1}{(t-1)z+1}\binom{tz}{(t-1)z}.$$

Substituting $t = m$ and $z = (mi-1)/(m-1)$, we have that the number of distinct unlabeled trees with i leaves is

$$\frac{1}{(m-1)(mi-1)/(m-1)+1}\binom{m(mi-1)/(m-1)}{(m-1)(mi-1)/(m-1)} = \frac{1}{mi}\binom{p(mi-1)}{mi-1}.$$

In a labeled m-ary tree over n variables, each leaf node is labeled with one of the m classes, and each internal node is assigned one of the n variables. Since an unlabeled m-ary tree on n variables with i leaves has $(i-1)/(m-1)$ internal nodes, it can be assigned at most $m^i n^{(i-1)/(m-1)}$ labelings. Hence, the number of distinct m-ary decision trees on n variables with at most k leaves is at most

$$\begin{aligned}
\sum_{i=1}^{k} \frac{m^i n^{(i-1)/(m-1)}}{mi}\binom{p(mi-1)}{mi-1} &< (mn)^k \sum_{i=1}^{k}\binom{p(mi-1)}{mi-1} \\
&\leq (mn)^k 2^{p(mk-1)} \\
&\leq (mn)^k 2^{2(mk-1)} \\
&< (4^m mn)^k.
\end{aligned}$$

Hence $|F_m(n,r)| \leq (4^m mn)^{(emn/r)^r}$ in this case. □

Note that Lemma 7 proves that in the interesting case—when $n > r$—the (logarithm of the size of the) set $F_m(n, r)$ is polynomial-sized in n for fixed m and r, and that the number of nodes in any decision tree of rank r is at most polynomial in the number of nodes in the smallest decision tree of rank r. The latter result implies that the representation length of any decision tree of rank r, using a suitable representation (e.g., the nested parentheses representation [Knuth 1969]), is polynomial in the representation length of the smallest decision tree of rank r.

Let us briefly consider the intuition behind the definition of rank and the motivation of learning decision trees of minimum rank. As already stated, learning decision trees that are optimal with respect to several characteristic measurements has turned out to be unfeasible in practice [Hyafil & Rivest 1976, Comer & Sethi 1977, Murphy & McCraw 1991] and it is generally believed that learning decision trees that fulfill any stringent optimality criterion is a NP-complete problem [Hancock *et al*. 1995]. The main motivation of Ehrenfeucht and Haussler [1989] in defining the rank of a decision tree has been to loosen the requirements of learning, but still retain some guarantees for the resulting classifier. According to the definition of rank, as demonstrated by the previous lemma, a tree of minimum rank is guaranteed to be within polynomial in size from the optimal one.

What does it mean in practice that a tree is of minimum rank? What is its relation to other possible representations of the underlying function? Blum [1992] has given one characterization while proving that decision trees with rank at most r are a subset of the concept class r-decision lists [Rivest 1987]. Clearly a decision tree of rank 1 is a special case of a decision list, which has single attribute tests in the conditions of rules. Blum went on to show that, for any r, a decision tree of rank r can be embedded into a decision list with at most r conjuncts in its condition terms. A side product of this construction is the following characterization for a decision tree of rank r: There always exists a path of length at most r from any internal node of the tree to a leaf. In particular, this holds for the root as well. More technical characterizations for decision trees of rank r can be found in Simon's [1991] paper.

We concentrate, next, on examining the learnability of decision trees that are of the smallest possible rank r for the given sample. Hence, the hypothesis is selected from among decision trees of rank r. With the implications of Lemma 7 at hand—agreeing on, say, the nested parentheses representation of decision trees—we put the decision tree representation issues aside for the remainder of this chapter.

3.2 Finding consistent decision trees of minimum rank

In this section we give a method for identifying a decision tree that is consistent with the given sample, prove its correctness, and analyze its computational complexity. Then, the procedure is used to construct an identification algorithm for decision trees of minimum rank.

Notation Let S be a sample of an n-ary function f on $[m]$ and v be a variable in $V_{n,m}$. Assume $v = v_i$, for some $1 \leq i \leq n$. Then S_k^v, where $k \in [m]$, denotes the set of examples $\langle x, f(x) \rangle$ in S such that $x = (x_1, \ldots, x_n)$ and $x_i = k$. We say v is *informative* (on S) if there exist distinct $i, j \in [m]$ such that both S_i^v and S_j^v are nonempty. The *rank* of a sample S, denoted by $r(S)$, is the minimum rank of any decision tree consistent with S.

The construction algorithm for decision trees of rank at most r follows the same recursive divide-and-conquer control structure as Procedure 2.3 *GrowTree*. However, since we this time need to ascertain the minimum rank of the resulting tree, we cannot do without backtracking. Given a sample S of a function and a rank bound r, Procedure 3.1, which is a strict generalization of Ehrenfeucht and Haussler's [1989] procedure *Find*, returns a decision tree of rank at most r consistent with the sample S, if one exists. Otherwise failure is reported.

The subprograms evoked by *Find* are as follows. It is simplest to consider *Exit* to be a macro, such that code "**return** T; **terminate**" will be expanded in place of the call *Exit*(T). In particular, it is the procedure *Find* that returns T and then terminates its execution. Similarly as in Section 2.1, *StoppingCondition* is a Boolean-valued function, which, here, returns **true** if all examples in S have the same class $k \in [m]$, otherwise **false** is returned (k is an out parameter). Function *MakeTree*, again, puts together a decision tree from its arguments so that the variable in its first parameter position will be the label of the tree's root and the following arguments will be the tree's subtrees.

First procedure *Find* evokes *StoppingCondition* to check whether all examples belong to the same class (Line 1); if successful, macro *Exit* is used to return an one-leaf decision tree and terminate the execution of *Find*. Otherwise, if the rank bound has value 0, failure has to be reported (Line 2). For larger rank bounds, an informative variable is attempted as the label of the root of the evolving tree (Line 3). If necessary, all informative variables are attempted in their turn. For all subsets of the sample determined by the chosen variable, the tree construction continues by recursively calling procedure *Find* (Line 4). In order to ascertain that the final tree has rank at most r, the recursive calls are evoked with reduced rank bound $r - 1$. Furthermore, the chosen variable becomes uninformative with respect to the subsets and can, therefore, be deleted from the set of available variables in the recursive calls. If all recursive calls are successful—i.e., return a consistent decision tree of rank at most $(r - 1)$—the final decision tree can be put together (Line 5). On the other hand, if a single call proves unsuccessful, then the definition of rank gives us the possibility to repeat that one recursive call with a higher rank bound, r, and still obtain a final tree of rank r (Lines 6–8). If the rerun call, however, is unsuccessful, then there cannot exist a decision tree of rank r for the given sample and failure has to be reported (Line 8). Finally, if all permutations of informative variables

Procedure 3.1 *Find*(S, r, V)
input: a nonempty sample S of some n-ary function f on $[m]$, an integer $r \geq 0$,
and a set of variables $V \subseteq V_{n,m}$.
output: a decision tree T of rank at most r that is consistent with S if one exists, else none.
begin
(1) **if** *StoppingCondition*(S, k) **then** *Exit*$(T = k)$ **fi**;
(2) **if** $r = 0$ **then** *Exit*(none) **fi**;
(3) **for each** informative variable $v \in V$ **do**
(4) **for each** $k \in [m]$ **do** $T_k^v \leftarrow$ *Find*$(S_k^v, r-1, V \setminus \{v\})$ **od**;
(5) **if** $\forall k \in [m] : T_k^v \neq \text{none}$ **then** $T \leftarrow$ *MakeTree*$(v, T_1^v, \ldots, T_m^v)$; *Exit*$(T)$ **fi**;
(6) **if** $T_k^v = \text{none}$ for a single value $k = \ell \in [m]$ **then**
(7) $T_\ell^v \leftarrow$ *Find*$(S_\ell^v, r, V \setminus \{v\})$;
(8) **if** $T_\ell^v \neq \text{none}$ **then** $T \leftarrow$ *MakeTree*$(v, T_1^v, \ldots, T_m^v)$ **else** $T \leftarrow \text{none}$ **fi**;
(9) *Exit*(T)
fi
od;
(10) *Exit*(none)
end.

have been attempted without success, failure has to be reported (Line 10).

There is a slight oversimplification in Ehrenfeucht and Haussler's [1989, p. 237] proof of their algorithm's correctness, when they state that: "If we stop in [Step] 3(c) [Line 9 in Procedure 3.1] returning "none," by the inductive hypothesis we must have either $r(S_0^v) > r$ or $r(S_1^v) > r$ for some variable v, and hence, since $S_0^v, S_1^v \subseteq S$, $r(S) > r$." Even though this deduction is valid, it is not immediate. The decision trees for the samples S_0^v and S_1^v depend on different attributes than the sample S: The variable v is uninformative for samples S_0^v and S_1^v, and can, thus, not appear in their decision trees, whereas a decision tree for sample S may contain tests for the variable v. If the sample S was first split by some other variable w (w would be assigned to the root of the tree), then maybe we could find eligible subtrees for S_0^w and S_1^w. That this, however, is not the case is shown explicitly in the following lemma:

Lemma 8 *Let S be a sample on n Boolean variables V_n. Let $r(S) = r$. Then each informative variable $v \in V_n$ that splits S so that either $r(S_0^v) \leq r-1$ or $r(S_1^v) \leq r-1$, has also $r(S_1^v) \leq r$ or $r(S_0^v) \leq r$, respectively.*

Proof Let v be the label of the root of a decision tree T of rank r that is consistent with S. Then T has subtrees of rank at most $r-1$ and r, by the definition of rank. Assume

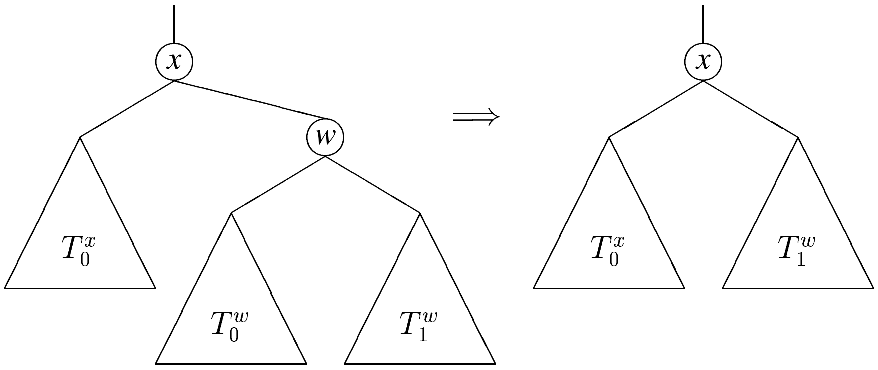


Figure 3.1: The pull-up operation in the case of binary attributes.

without loss of generality that the 0-subtree T_0^v has rank $r - 1$ and that the 1-subtree T_1^v has rank r.

Now, let w be another variable such that it is informative for S and splits S to subsets so that the rank of the other is at most $r - 1$. Assume without loss of generality that $r(S_0^w) \leq r - 1$. We can now construct the 1-subtree for sample S_1^w from the decision tree T. There are two cases:

1. w is tested in tree T. Then deleting from T the nodes that are labeled with w and those nodes' 0-subtrees clearly results in a valid decision tree T' of rank at most $r(T) = r$ consistent with S_1^w. The 1-subtrees of the deleted nodes are pulled up to replace the removed parts (see Fig. 3.1).

2. w does not appear in T. In this case T can be used as such as the 1-subtree of the tree.

In both cases a 1-subtree of rank at most r could be constructed from T. Hence, $r(S_1^w) \leq r$. □

The above lemma is presented for the binary case for simplicity. It generalizes to multivalued attributes in a straightforward way: If a m-valued root node v has $m - 1$ subtrees of rank at most $r - 1$ for a sample S of rank r, then the remaining subtree, denoted k-subtree, has rank at most r. In the proof of the lemma, the remaining subtree, k-subtree, can be constructed by deleting from T all nodes labeled with v and all their subtrees, except k-subtrees, similarly as in the Boolean case.

The proofs of the correctness and the running time of the procedure are simple variations of those for the Boolean case [Ehrenfeucht & Haussler 1989].

Lemma 9 *The procedure Find is correct.*

Proof The correctness is established by induction on $|S|$ and r.

If $|S| = 1$ or $r = 0$ then it is clear that the procedure *Find*($S, r, V_{n,m}$) is correct.

Now assume S is a sample with $|S| = h \geq 2$ and $r \geq 1$. Assume that the procedure is correct for $r - 1$ when S has arbitrary size, and for r when S has size less than h.

For any informative variable v, $|S_k^v| < |S|$ for all $k \in [m]$. Thus, it is clear that if *Find*($S, r, V_{n,m}$) does return a tree then by the inductive hypothesis and the definition of rank it will be a tree of rank at most r that is consistent with S.

If, on the other hand, none is reported, then, since $r \geq 1$, execution must stop either on Line 9 or Line 10. If we stop on Line 9 returning none, by the inductive hypothesis we must have $r(S_k^v) > r$ for some $k \in [m]$ and $r(S_i^v) \leq r - 1$ for all $i \neq k$ for some informative variable v, and hence, by the generalization of Lemma 8, $r(S) > r$. If we stop on Line 10, by the inductive hypothesis, we must have, for at least two distinct values $i, j \in [m]$, $r(S_i^v), r(S_j^v) \geq r$ for every informative variable v. Now, let T be a decision tree of rank $r(S)$ that is consistent with S and has a minimal number of nodes. The root of T must be labeled with a variable v that is informative for S, for otherwise we can find a smaller decision tree of rank $r(S)$ consistent with S. Now assume that $i, j \in [m]$ are the two distinct values for which $r(S_i^v), r(S_j^v) \geq r$. The i-subtree of T must be consistent with S_i^v and the j-subtree with S_j^v. Hence at least these two subtrees must have rank at least r, and therefore $r(T) > r$ by the definition of rank. Thus $r(S) > r$. Hence in any case the procedure *Find* is correct. □

In the following we prove that Procedure 3.1 *Find* is efficient in the sense that its running time is only linear with respect to the size of the sample and polynomial with respect to the number of variables, n, and their arity, m, for fixed rank bound r.

Lemma 10 *For any nonempty sample S of an n-ary function f on $[m]$ and $r \geq 0$, the time of Find(S, r) is $O(|S|(m/2)^r(n+1)^{2r})$.*

Proof Fix $n \geq 0$ and $k \geq 1$, and let $T(i, r)$ be the maximum time needed for *Find*($S, r, V_{n,m}$) when S is a sample of a function $f\colon [m]^n \to [m]$ with $1 \leq |S| \leq k$ and at most i variables are informative on S.

If $i = 0$ then $T(i, r)$ is $O(1)$, since $|S| = 1$ in this case. If $r = 0$ then $T(i, r)$ is clearly $O(k)$. If $r \geq 1$ then the time required to test if all examples are of the same class (function *StoppingCondition*), and to determine which variables are informative (Line 3), and to perform other miscellaneous tests in the procedure is $O(kn)$. Each recursive call on Line 4 takes time at most $T(i - 1, r - 1)$ since the variable v is no longer informative in S_j^v for any $j \in [m]$. These calls are made at most i times in the course of the loop on Lines 3–9, yielding thus a total time for all executions on Line 4 of at most $miT(i-1, r-1)$. The only remaining action is on Line 7, where a recursive call is made to *Find*($S_j^v, r, V \setminus \{v\}$) for some $j \in [m]$ for some informative variable v. This takes time at most $T(i-1,r)$. Since the conditional statement on Lines 6–9 terminates the loop, this call is made at most once. It follows that for $r \geq 1$,

$$T(i,r) \leq O(kn) + miT(i-1,r-1) + T(i-1,r).$$

Thus we have the following recurrence for $T(i,r)$:

$$\begin{aligned} &T(0,r) \leq c_1 \text{ for all } r \geq 0, \\ &T(i,0) \leq c_1 \text{ for all } i \geq 0, \text{ and} \\ &T(i,r) \leq c_2 + miT(i-1,r-1) + T(i-1,r) \text{ for all } i,r \geq 1, \end{aligned}$$

where c_1 and c_2 are positive constants that are $O(k)$ and $O(kn)$, respectively. Solving the last term, it follows that

$$\begin{aligned} T(i,r) &\leq c_2 + miT(i-1,r-1) + \sum_{j=1}^{i-1}(c_2 + mjT(j-1,r-1)) + T(0,r) \\ &\leq c_2 i + m\sum_{j=1}^{i} jT(j-1,r-1) + c_1 \\ &\leq c_1 + c_2 i + m\frac{i(i+1)}{2}T(i,r-1), \end{aligned}$$

since $T(i,r)$ is clearly an increasing function in the range of i. Hence

$$T(i,r) < c_1 + c_2(i+1) + \frac{1}{2}m(i+1)^2 T(i,r-1).$$

Solving, it follows that

$$\begin{aligned} T(i,r) &\leq c_2\sum_{j=0}^{r-1}\left(\frac{m}{2}\right)^j (i+1)^{2j+1} + c_1\sum_{j=0}^{r}\left(\frac{m}{2}\right)^j (i+1)^{2j} \\ &\leq O\left(kn\left(\frac{m}{2}\right)^{r-1}(i+1)^{2r-1} + k\left(\frac{m}{2}\right)^r (i+1)^{2r}\right). \end{aligned}$$

Since $i \leq n$ and $k = |S|$, this implies that the time for *Find*($S, r, V_{n,m}$) is

$$O(|S|(m/2)^r(n+1)^{2r}). \qquad \square$$

Given the procedure *Find*, we can now construct an algorithm *Findmin*(S) to find a minimum rank decision tree for a sample S by simply executing *Find*($S, r, V_{n,m}$) for consecutive rank candidates $r = 0, 1, 2, \ldots$ until a decision tree is returned. By Lemma 10, the time for *Findmin*(S) is

$$O\left(\sum_{r=0}^{r(S)} O(|S|(m/2)^r(n+1)^{2r})\right) = O(|S|m^{r(S)}(n+1)^{2r(S)}).$$

Hence we have

Theorem 11 *Given a sample S of an n-ary function f on $[m]$, using Findmin(S) we can produce a decision tree that is consistent with S and has rank $r(S)$ in time*

$$O(|S|m^{r(S)}(n+1)^{2r(S)}). \qquad \square$$

Lemma 7 and Theorem 11 have already given us the results that would be needed to conclude the learnability of decision trees of bounded rank if the function classes $F_m(n,r)$ were representation length bounded projections (cf. [Natarajan 1991]). As it happens, there is some overlap between these function classes, as can be observed from Lemma 7. That is, the minimum rank decision tree for a function f is not necessarily the smallest (and shortest) representation of f. Therefore, we use Corollary 3 explicitly to conclude the learnability of decision trees of at most a fixed rank.

Theorem 12 *For any $n \geq r \geq 1$ and $m \geq 2$, any target function $f \in F_m(n,r)$, any probability distribution P on $[m]^n$ and any $0 < \varepsilon$, $\delta < 1$, given a sample S derived from a sequence of at least*

$$\frac{1}{\varepsilon}\left(\left(\frac{emn}{r}\right)^r \ln(4^m mn) + \ln\frac{1}{\delta}\right)$$

random examples of f chosen independently according to P, with probability at least $1-\delta$, Find($S, r, V_{n,m}$) (resp. Findmin(S)) produces a hypothesis $g \in F_m(n,r)$ that has error at most ε.

Proof By Lemma 7, $|F_m(n,r)| \leq (4^m mn)^{(emn/r)^r}$ for $n \geq r \geq 1$. Hence by Corollary 3, with probability at least $1-\delta$, every hypothesis $g \in F_m(n,r)$ that is consistent with a sequence of

$$\begin{aligned} \frac{1}{\varepsilon}\ln\frac{|F_m(n,r)|}{\delta} &\leq \frac{1}{\varepsilon}\ln\frac{(4^m mn)^{(emn/r)^r}}{\delta} \\ &= \frac{1}{\varepsilon}\left(\left(\frac{emn}{r}\right)^r \ln(4^m mn) + \ln\frac{1}{\delta}\right) \\ &\leq |S| \end{aligned}$$

random examples of f has error at most ε. Since *Find*($S, r, V_{n,m}$) and *Findmin*(S) both produce one of these hypotheses, the result follows. $\square$

The preceding results show that m-ary decision trees of rank at most r on n variables can be learned with accuracy $1-\varepsilon$ and confidence $1-\delta$ in time $O((n^{O(r)}/\varepsilon)\log(1/\delta))$. Since this is polynomial in $1/\varepsilon$, $1/\delta$, and n for fixed r, and since the nested parentheses representation length of a decision tree of rank r is at most polynomial in the length of the nested parentheses representation of any other decision tree of rank r, this implies that m-ary decision trees of rank at most r are polynomially learnable.

Theorem 13 *The class of n-ary functions represented by m-ary decision trees of rank at most r, where $m, n, r \in \mathbb{N}$, $n > r \geq 1$ and $m \geq 2$, is polynomially learnable.* $\square$

3.3 Incremental construction of minimum rank decision trees

In this section we show how a decision tree of minimum rank can be constructed in the incremental setting, where the examples are received one at a time and the objective is to maintain a consistent decision tree of minimum rank by doing as few modifications to the existing hypothesis as possible. For a sequence S of examples the method that we develop is shown to require (asymptotically) at most the same total amount of time as *Findmin*, which is presented with the whole sample S at once.

3.3.1 Updating a hypothesis efficiently

Before motivating the incremental version of *Findmin*, it is worth taking notice of the following simple observation, which, together with Lemma 8, plays an important role in the subsequent results.

Lemma 14 $r(S) \leq r(S \cup \{e\}) \leq r(S) + 1$ *for all example sets* S *and examples* e*, where* $S \cup \{e\}$ *is consistent.*

Proof First, assume that, on the contrary, $r(S \cup \{e\}) < r(S)$. Then there exists a decision tree T of rank strictly less than $r(S)$ that is consistent with $S \cup \{e\}$. But T is also consistent with any subset of $S \cup \{e\}$, in particular with S too. We have a contradiction with the assumption. Hence, the first inequality holds.

For the second inequality, we show that from a decision tree T of rank $r(S)$, which is consistent with S, we can always construct a decision tree of rank at most $r(S) + 1$ that is consistent with $S \cup \{e\}$. There are two cases:

1. The example e is consistent with T. In this case there is no need to update the hypothesis; T is a consistent decision tree of the minimum rank $r(S)$.

2. The example e is inconsistent with T. Then we can prove the claim by induction over $|S|$.

   For the case $|S| = 0$ the value $r(S)$ is undefined. When $|S| = 1$ the rank of S is invariably 0 and the rank of $S \cup \{e\}$ is invariably 1. Thus, the second inequality holds in this case.

   Let us now make the inductive hypothesis that the claim holds for all values $|S| < k$ and let us then consider the situation $|S| = k$.

   Since all examples are not of the same class, it has to be that the decision tree Q of minimum rank that is consistent with $S \cup \{e\}$ has a root node labeled with an informative attribute; i.e., the root node partitions $S \cup \{e\}$ into at least two disjoint nonempty subsets S_1 and S_2. Let us assume, without loss of generality, that $e \in S_1$. Since, both S_1 and S_2 are nonempty, it must be that $1 \leq |S_1| \leq |S|$. Now $r(S_1) = r(S'_1 \cup \{e\})$, where $|S'_1| < |S| = k$, hence, by the inductive hypothesis, $r(S_1) \leq r(S) + 1$. Furthermore, $r(S_i) \leq r(S)$ for all other values $i \neq 1$ by the first inequality, because $S_i \subseteq S$. Hence, by the definition of rank, the rank of the tree Q is at most $r(S) + 1$.

□

Now we give the rationale for the incremental version of the *Findmin* algorithm. For clarity, we present the reasoning in the binary setting; its generalization to multivalued case should be clear.

In the following let S be the sample of size q observed thus far and let T be a decision tree of rank $r(S)$ that is consistent with S. In addition, let e be a new observation that is consistent with S. Our objective here is to update T to classify correctly e together with S. We want only few, if any, changes to the tree T to happen and require that the modified tree is of minimum rank for the extended sample $S \cup \{e\}$, i.e., its rank increases (by one) only if $r(S \cup \{e\}) = r(S) + 1$. If N is a node in a decision tree Q, then by Q_N we denote the subtree of Q that is rooted at N. Sometimes we talk about the rank of a node N, formally we mean the rank of Q_N. By $S_N \subseteq S$ we denote the examples that are associated with (the leaves of) the subtree Q_N.

Directing e down the tree T will result in e at one of the leaf nodes of T. Let us denote that leaf by L. Only if the label in L differs from that of e, does T need to be modified. Note that, by Lemma 14, there always exists a decision tree T_1 of rank 1 that is consistent with $S_L \cup \{e\}$. In some cases substituting the subtree T_1 for L in T will not increase the rank of the tree T. Then we know that the modified decision tree is consistent with and, by Lemma 14, of minimum rank for $S \cup \{e\}$. Let us now review these cases.

Let P be the parent node (if any) of L and let K be the sibling of L (see Fig. 3.2). The subtree T_P rooted at P has rank at least 1. If $r(T_P) > 1$, then, by the definition of rank, it must be that $r(T_K) = r(T_P) > 1$, and increasing the rank of the subtree rooted at L from 0 to 1 will not change the rank of T_P and, thus, will not affect the rank of the whole tree T. In this case, the subtree T_1 may safely be substituted for the leaf L.

The remaining possibility is that $r(T_P) = 1$ and it has two subcases: $r(T_K)$ can now be either 0 or 1. If $r(T_K) = 0$, we can still increase the rank of the subtree rooted at L freely, since $r(T_P)$ will not change in this case. If, on the other hand, $r(T_K) = 1$, increasing the value of $r(T_L)$ to 1 would now increase $r(T_P)$ and potentially propagate to affect the rank of T. However, that is not necessarily the case; there may be an ancestor node A of L (on the path to the root) such that its rank may safely be incremented by one without affecting the rank of the whole tree T. If such a *safe* ancestor exists, all nodes in the (sub)tree T_A inherit the property. Therefore, it suffices to test the property for the node L.

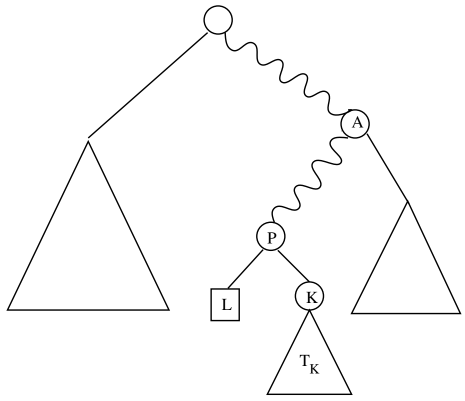


Figure 3.2: When directed down the decision tree, example e ends up in leaf node L. Node P is the parent of L and K is the sibling of L.

Finally, it may be that L is neither safe in its own right nor because of a safe ancestor. Even then there still is a possibility of finding a consistent tree of rank $r(S)$ for the sample $S \cup \{e\}$. We do not have to start to search for such a tree from the scratch; we can reap reward of the work already done in constructing T. Note that if a decision tree was not consistent with S, neither will it be with $S \cup \{e\}$; the same holds for subsamples. Therefore, if a (sub)tree was rejected as the hypothesis of (a subset of) S, we do not have to reconsider it in connection of $S \cup \{e\}$.

Starting from the parent P of L we do a bottom-up traversal on the path from the root to L. At each node en route we search for a decision tree of the same rank that the node currently has. The search begins from the first (informative) attribute that has not been considered previously. Attributes, which were previously uninformative, have to be reconsidered in order to determine whether the addition of example e has changed their status.

The traversal terminates when a consistent subtree of the current rank is found at some level or when the root has been processed without success. In case of the first termination condition we substitute the newly found subtree for the old one and, thus, create a consistent tree of rank $r(S)$ for $S \cup \{e\}$. In the latter case, all trees of rank $r(S)$ have been considered and none of them has proven to be consistent with $S \cup \{e\}$ (or [previously] with a subset of it). Therefore, it must be that $r(S \cup \{e\}) = r(S) + 1$. We are already aware of one consistent decision tree of minimum rank for $S \cup \{e\}$, viz., the initial tree T with subtree T_1 substituted for T_L is such a hypothesis.

In some nodes there is no need to continue the search for the uncovered variables. Consider the situation where the left subtree, in which the example e falls into, of ancestor node A in Fig. 3.2 has rank $r(T_A)$ (in that case the right subtree has rank $r(T_A) - 1$ or

Table 3.1: Possible situations and the corresponding update actions taken when a new example is directed to the leaf L in the left subtree of the current hypothesis of rank r.

<table>
<tr><th rowspan="2">SITUATION TYPE</th><th colspan="2">SUBTREES</th><th rowspan="2">UPDATE ACTION</th></tr>
<tr><th>left</th><th>right</th></tr>
<tr><td>1: SAFE</td><td>$r-1$</td><td>$r-1$</td><td rowspan="2">Substitute T_1 for T_L</td></tr>
<tr><td>2: SAFE</td><td>$< r-1$</td><td>r</td></tr>
<tr><td>3</td><td>$r-1$</td><td>r</td><td rowspan="2">If none of the nodes on the path from the root to the leaf L is of type 1 or 2, then start to search for a tree of the current rank from the parent of L. Skip the exhausted nodes en route.</td></tr>
<tr><td>4: EXHAUSTED</td><td>r</td><td>$\leq r-1$</td></tr>
</table>

less). Then, by Lemma 8, if the current variable in the node leads to a left subtree of rank $r(T_A)+1$, the example set $S_A \cup \{e\}$ has rank $r(T_A)+1$. Therefore, there is no point in continuing the search for a tree of lesser rank for that example set. The search continues at the parent node of A for a larger example set. In this case we say that A is *exhausted* (with respect to e). In particular, at the root level this means that if the hypothesis has subtrees of different rank and the disagreeing example is directed to the one with larger rank, then if the search returns unsuccessfully to the root, there does not exist a decision tree of the same rank such that it is consistent with the extended sample.

Let us reiterate and consider the required update operations at the root level (cf. Table 3.1).

1. If the root node is safe, i.e.,

   (a) if the subtrees have equal ranks or

   (b) there is a difference of more than 1 in their ranks and the example falls into the one with lesser rank, then

   it suffices to construct a decision tree T_1 of rank 1 for the subsample $S_L \cup \{e\}$ and substitute it for the leaf L in the tree.

2. Otherwise, a bottom-up traversal on the path from the root to the leaf L needs to be performed. At each node a search for a subtree of the current rank is carried out for those variables that have not been tried yet. When the search reaches the root without success, then

   (a) if the example fell into the subtree that has rank smaller by one than that of the other subtree (Situation 3), a new root variable has to be searched for from among those not considered yet. If none of them yields the result, the rank of the sample has increased, and it is enough to substitute T_1 for L in the tree.

(b) If, on the other hand, the example fell to the subtree that has the higher (by one) rank, the root is exhausted with respect to the example and the trivial subtree substitution suffices.

Note in particular that only one out of the four situations above can lead to changing the variable at the root, i.e., discarding the current hypothesis completely. In the safe cases a trivial replacement of a leaf needs to be performed and in the potentially exhausted case, at worst, one of the subtrees is rebuilt from the scratch. From Table 3.1 we can clearly see that the number of tree structures that may lead to updates that discard the existing hypothesis (Situation 3) is small compared to the total number of tree structures, since they only comprise a part of the exhausted ones, which together with the safe structures do not require total reconstruction.

The method described above meets the requirements that were initially presented to a successful incremental learner: It modifies the existing decision tree as little as possible only when necessary and increases the rank of the hypothesis only if the rank of the sample grows because of the addition of a new instance. Next we present the method in more algorithmic form.

3.3.2 The incremental algorithm

In order to realize the incremental version of *Findmin*, as motivated above, we need to attach some bookkeeping information to the hypothesis. Each node N (subtree T_N) of the hypothesis tree T will have two sets associated with it: *Exs*(N) is the set of examples that fall into (the leaves of) subtree T_N and the set *Vars*(N) consists of the informative variables that have not been evaluated in this node (since the last change in an ancestor node) and of the uninformative variables (which may become informative later).

Procedure *Find* is modified in this task only to include some added bookkeeping operations and a changed *Exit* macro. The procedure *StoppingCondition* remains as it is. The bookkeeping operations are embedded in the auxiliary subprograms: *Initialize* sets the global variable W to value V (Line 2′) and procedure *Update* deletes the selected variable v from set W on Line 3′ of *Find*, i.e., $W \leftarrow W \setminus \{v\}$. Most of the bookkeeping is carried out when exiting the procedure *Find*. Macro 3.3 *Exit* updates the values of the sets associated with a node: The set *Exs* associated with the root of the returned tree always contains the whole sample S; the update of the set *Vars* depends on the number of the call, which now is the second parameter of the macro. When a leaf node (call number 1) is returned or if a tree is found without having to build a subtree of rank r (call number 3), the unevaluated variables can be obtained from the set W. When the search proves unsuccessful (call numbers 2 and 5), updating *Vars* is rendered superfluous. If a subtree

---

Procedure 3.2 *Find*(S, r, V)
input: a nonempty sample S of some n-ary function f on $[m]$, an integer $r \geq 0$,
and a set of variables $V \subseteq V_{n,m}$.
output: a decision tree T of rank at most r that is consistent with S if one exists, else none.
begin
(1) **if** *StoppingCondition*(S, k) **then** *Exit*$(1, T = k)$ **fi**;
(2) **if** $r = 0$ **then** *Exit*$(2, \text{none})$ **fi**;
(2′) *Initialize*;
(3) **for each** informative variable $v \in V$ **do**
(3′) *Update*;
(4) **for each** $k \in [m]$ **do** $T_k^v \leftarrow$ *Find*$(S_k^v, r-1, V \setminus \{v\})$ **od**;
(5) **if** $\forall k \in [m] : T_k^v \neq \text{none}$ **then** $T \leftarrow$ *MakeTree*$(v, T_1^v, \ldots, T_m^v)$; *Exit*$(3, T)$ **fi**;
(6) **if** $T_k^v = \text{none}$ for a single value $k = \ell \in [m]$ **then**
(7) $T_\ell^v \leftarrow$ *Find*$(S_\ell^v, r, V \setminus \{v\})$;
(8) **if** $T_\ell^v \neq \text{none}$ **then** $T \leftarrow$ *MakeTree*$(v, T_1^v, \ldots, T_m^v)$ **else** $T \leftarrow \text{none}$ **fi**;
(9) *Exit*$(4, T)$
fi
od;
(10) *Exit*$(5, \text{none})$
end.

---

of rank r has been successfully built (call number 4), then all variables can be marked evaluated, since, because of Lemma 8, there is no point in maintaining any variables as possible continuation points of the search.

The higher level search control in the incremental algorithm has to emulate that of *Findmin* by maintaining, between examples, the intermediate states that *Findmin* comes across during its search procedure. The incremental main program is called *IFM*. It uses a stack to control path traversal. Standard stack operations are evoked by *IFM*: Function *Pop* returns the top element of the stack if one exists; function *Top* tells whether the stack is empty or not. Call *Substitute*(T_1, T_2) replaces the (sub)tree T_2 by the tree T_1.

The algorithm *IFM* first uses the existing hypothesis T to classify the new example e (Lines 1–4). The example sets associated with the nodes on the path are updated simultaneously. Once the leaf L, where e ends up in, has been found, *IFM* evokes *Find* to construct a tree T_H of rank at most 1 for the sample $S_L \cup \{e\}$ (Line 5) and makes a copy T' of T in which leaf L is replaced by subtree T_H (Line 6). If e is consistent with T, then *Find* returns another leaf and the rank of T' remains unchanged. Even if e disagrees with T, but L is a safe node, the rank of the modified copy T' remains unchanged, and it can be returned as the updated hypothesis (Line 12). Otherwise, we continue the search

---

Macro 3.3 *Exit*(n, T)
input: an integer $n \in [5]$ and a decision tree T.
begin
(1) $\quad Exs(T) \leftarrow S$;
(2) $\quad$ **case** n **is**
(3) $\qquad 1, 3$: $Vars(T) \leftarrow W$;
(4) $\qquad 2, 5$: **null**; $\qquad$ % do nothing
(5) $\qquad 4$: $Vars(T) \leftarrow \emptyset$
$\quad$ **end**;
(6) $\quad$ **return** T; **terminate** $\qquad$ % the execution of the calling procedure
end.

---

from the parent of L (Lines 8–11): A new subtree of the current rank is searched for using the variables that have not been tried before. The loop terminates when successful or when the root has been processed without success, in which case the new hypothesis constructed on Line 6 will be returned even though it has increased rank. In both cases the variable sets associated with the nodes of the tree need to be updated. Notice that unnecessary searching in case of exhausted nodes is avoided through careful control of the variable sets associated with the nodes of the tree.

The correctness of *IFM* should be clear from the rationale presented above.

Theorem 15 *Let T be a decision tree of rank $r(S)$ that is consistent with sample S of an n-ary function on $[m]$ and let e be an example of the same function. Then the call $IFM(T, e)$ returns a decision tree that is consistent with $S \cup \{e\}$ and has rank $r(S \cup \{e\})$.*

Proof Follows almost directly from the correctness of *Find* (Lemma 9). There is no elegant way to prove *IFM*'s correctness. The proof proceeds in a similar fashion as the rationale of the algorithm. Since rigorous motivation and verbal analysis of the situation was already presented, we pass over the formal presentation of the proof. $\square$

To denote repeating application of *IFM* to the members of an example sequence S we write *IFM*(S) = *IFM*(*IFM*$(\ldots($*IFM*$(\perp, s_1)\ldots), s_{q-1}), s_q)$, where $S = \langle s_1, \ldots, s_q \rangle$ and $\perp$ denotes the empty hypothesis. Incidentally, note that the result of call *IFM*(S) is not uniquely determined; i.e., *IFM* is not invariant to the permutation of an example sequence, but the result depends on the order of examples in that sequence. Clearly, for instance, different attributes are retained in the variable sets associated with the nodes of the evolving tree depending on the examples seen. This, then, means also that the trees

Algorithm 3.4 *IFM*(T, e)
input: a decision tree T of rank $r(S)$ that is consistent with a sample S of some n-ary function on $[m]$ and an example e of the same function.
output: a decision tree T' of rank $r(S \cup \{e\})$ that is consistent with $S \cup \{e\}$.
begin
Direct e down the tree T;
for each node N en route **do**
  $Exs(N) \leftarrow Exs(N) \cup \{e\}$;
  $Push(N)$
od;
$L \leftarrow Pop; T_H \leftarrow Find(Exs(L), 1, Vars(T_L))$;
$T' \leftarrow T$; *Substitute*(T_H, T'_L);
if $r(T) \neq r(T')$ **then** % $r(T_H) > 0$ and L was not a safe node
  $T_H \leftarrow$ none;
  while *Top* **and** $T_H =$ none **do**
    $L \leftarrow Pop; T_H \leftarrow Find(Exs(L), r(T_L), Vars(T_L))$;
    if $T_H \neq$ none **then** *Substitute*(T_H, T'_L) **fi**
  od
fi;
return T'
end.

produced by *Findmin* and *IFM* are not necessarily the same. What remains invariant to the permutation is the rank of the resulting tree.

IFM explicitly simulates the functioning of the nested recursive calls to *Find* evoked by *Findmin* and, hence, does exactly the same amount of work in searching for a consistent concept for a sequence of examples that would be carried out by *Findmin* for the set consisting of the elements of that sequence. Therefore, we have the same asymptotic time complexities for *IFM*(S) and *Findmin*(S). (In addition *IFM* does some bookkeeping, which does not affect the asymptotic time requirement.)

Theorem 16 *Given an sample S of an n-ary function on $[m]$, using IFM(S) we can produce a decision tree that is consistent with S and has rank $r(S)$ in time $O(|S|m^{r(S)}(n+1)^{2r(S)})$.*

Proof The proof is essentially based on the time requirement of *Find* and on the fact that no more recursive calls of *Find* will be made by *IFM* over a sequence S of examples than would happen during the execution of call *Findmin*(S). For the details of the proof see Appendix A. □

In the worst case a single call *IFM*(T, e) will take time $O((|S_T| + 1)m^{r(T)}n^{2r(T)})$. In other words, the time of an update grows as the length of the example sequence and the size of the hypothesis grow. However, as our earlier discussion shows, only few mistakes lead to time-consuming updates; updates where a simple subtree replacement is performed are carried out in linear time. Furthermore, even though time proportional to $n^{2r(T)}$ has to be spent on updates while the hypothesis has rank $r(T)$, before the rank of the hypothesis is incremented, that time is usually divided between several updates.

If *Findmin* was trivially used to generate a new hypothesis from the scratch after each disagreeing example then, in the worst case, total time

$$\sum_{i=1}^{q} \left(|S_i| m^{r(S_i)} (n+1)^{2r(S_i)}\right) = O\left(|S|^2 m^{r(S)+1} (n+1)^{2r(S)+1}\right)$$

would be needed for a sequence of q examples. Hence, using *IFM* instead of simply repeating *Findmin* for all the examples after each new example received over a span of time proves profitable.

Because of Theorems 15 and 16 all the PAC learning results and their consequences [Ehrenfeucht & Haussler 1989] hold for *IFM* as well as for *Findmin*.

3.3.3 On the number of erroneous predictions

Let us conclude this section by taking a little detour from our main theme—development of a practical decision tree learner—and briefly consider how *IFM* relates to the on-line learning model [Littlestone 1988] by giving an upper bound for the number of false predictions it will make on a sequence of examples. We continue to work within the Boolean world.

How often does *IFM* have to update its hypothesis? First, recall that only disagreeing examples initiate any changes to the hypothesis. On the other hand, they necessitate some changes. Hence, every disagreeing example causes an update of some severity. From the point of view of the on-line prediction a disagreeing example is a mistake made by the decision tree. Littlestone [1988] has studied absolute mistake bounds of on-line learning, i.e., what is the minimum number of unavoidable erroneous predictions in learning some concept class. In the following we apply that line of analysis to learning rank-bounded decision trees, even though the situation there is slightly different from that studied by Littlestone. The aim to maintain a consistent tree of the minimum rank at each stage requires that the target concept class changes dynamically.

The following lemma shows that *IFM* is a “halving algorithm” [Littlestone 1988] for $\mathrm{DT}_2^r(n)$. This follows from the fact that rank is a structure delimiting property with no stand to the semantics of the tree; i.e., it does not consider the function represented by the decision tree in any other respect except that it has to be consistent with the sample.

Lemma 17 *Let S be a sample of some Boolean function on $V_{n,2}$ such that $r(S) = r$. Then, IFM(S) makes at most $\log_2 |F_2(n, r)|$ mistakes.*

Proof Observe that for all $T \in \mathrm{DT}_2^r(n)$, where T is consistent with the example e, the set $\mathrm{DT}_2^r(n)$ also contains the inconsistent tree T_f that is equal to T in its structure but has the label of one leaf (the one that e ends in) flipped. Because of this duality each mistake at least halves the number of candidate functions that could be chosen as the hypothesis and the claim follows. □

Simon [1995] has studied learning rank bounded decision trees with equivalence queries [Angluin 1988] which is (more or less) equivalent to learning in the on-line setting: Instances are received one at a time and *reinforcement* (information regarding whether the instance was classified correctly or not) is provided concerning each instance [Littlestone 1988]. Simon presents an algorithm for learning decision trees of bounded rank efficiently with equivalence queries. The algorithm is similar to *IFM* in the sense that it, too, represents its hypotheses as decision trees. However, Simon's algorithm is far more general than *IFM* and does not attain the same efficiency.

It is easy to see that a decision tree of rank r has at least 2^r leaves. In other words, a sample that has rank r must contain at least 2^r examples. Hence, when the length of the example sequence is in between 2^r and 2^{r+1}, not even a disagreeing example can cause the rank of the hypothesis to increase from r to $r + 1$. Over a sequence S, since $|S| \geq 2^{r(S)}$, it must be that $r(S) \leq \lfloor \log_2 |S| \rfloor$.

3.4 Learning decision trees in the presence of noise

Elomaa and Kivinen [1991] and Sakakibara [1993] have independently demonstrated how the (binary) *Find* procedure is made robust against random classification noise. The technique they use is essentially the one used to derive procedure *GrowPrunedTree* from *GrowTree* in Section 2.1. This way they are able to prove the learnability of decision trees of bounded rank in the presence of random classification noise. In this section we show how the same techniques can be used in the extended learning model.

Before going any further we point out that, as our discussion in Section 2.1 would suggest, pruning lets the learning method in practice tolerate, not only random classification noise, but other types of noise too. The basic results of Elomaa and Kivinen [1991] and Sakakibara [1993] deal with the noise model of Angluin and Laird [1988]. Sakakibara further discusses how these results relate to learnability in Valiant's [1985] *malicious error model*. Bearing this in mind we only discuss random classification noise in the following.

The main theme of this section is to show the learnability of multiconcept classifiers in the presence of random classification noise. It suffices to show that Algorithm 3.5

Algorithm 3.5 *Rodt*$(S, r, n, \eta_b, \varepsilon, \delta)$
input: a nonempty noisy sample S of some n-ary function on $[m]$, integers $r \geq 0$ and $n > 0$, and positive reals η_b, ε, and δ, such that $\eta \leq \eta_b < 1/2$ and $0 < \varepsilon, \delta \leq 1$.
output: a decision tree of rank at most r.
begin
(1) $\kappa \leftarrow \left\lfloor \frac{\varepsilon(1-2\eta_b)|S|}{8L(n,r)} \right\rfloor$; $\gamma \leftarrow \eta_b + \frac{\varepsilon(1-2\eta_b)}{8}$;
(2) $T \leftarrow$ *Find*$(S, r, V_{n,m})$;
(3) **if** $T \neq$ none **then return** T **else return** an arbitrary decision tree of rank r **fi**
end.

Procedure 3.5 *StoppingCondition*(S)
input: a nonempty noisy sample S of some n-ary function on $[m]$.
output: a truth value.
begin
% Let M_i, $i = 1, \ldots, m$, be the number of examples in S labeled by i.
(1) **if** $M_k \geq \gamma|S|$ **or** $M_j \leq \kappa$ for all $j \neq k$ (and $M_k > \kappa$) **then return true else return false fi**
end.

Rodt—yet another variant of Ehrenfeucht and Haussler's method—is a noise-tolerant Occam algorithm for decision trees of bounded rank.

The main modification to procedure *Find* is to relax the fitting of the hypothesis to the sample as explained in Section 2.1. In growing the tree, we do not split the example set, if the number of examples in the set is less than a threshold κ or if at least a proportion γ of the examples already have the same label. Thus the candidate variable is pruned.

Now, in addition to the procedure *Find*, we need a main program that calculates the appropriate values of κ and γ, and calls *Find* with these values. The function $L(n, r)$ in the following is the maximum number of leaves in a decision tree from the proof of Lemma 7.

We prove that *Rodt* is a noise-tolerant Occam algorithm for m-ary decision trees of fixed rank by modifying the corresponding proof of Elomaa and Kivinen [1991]. They go about the task by considering rules induced by a decision tree.

A *rule* over the variables $v_1, \ldots, v_n \in V_{n,m}$ is a pair $\langle c, l \rangle$, where c is **true**, **false**, or a conjunction over assertions on the values of variables v_i, and l belongs to $[m]$. A rule $\langle c', l' \rangle$ is a *refinement* of the rule $\langle c, l \rangle$ if $l' = l$ and c' logically implies c. We say that an example $\langle x, l \rangle$ *matches* a rule $\langle c, l' \rangle$ if x satisfies c. If the example matches the rule and $l \neq l'$, we say that the example *disagrees* with the rule.

For a decision tree T we define the set $\varrho(T)$ of rules *induced by* T. Informally, each path leading from the root of the tree to a leaf induces one rule $\langle c, l\rangle$. The label l is obtained from the leaf. The conjunction c concerns exactly the variables appearing at the nodes on the path. The assertion concerning variable v in c is $(v = k)$ if the path leads to the k-th son of the node labeled by v. Thus we have the following recursive definition:

1. If T is a leaf labeled by l, then $\varrho(T) = \{\langle \mathbf{true}, l\rangle\}$.

2. If the root of T is labeled by v and T_i for $i = 1, \ldots, m$ are the subtrees of T, then

$$\varrho(T) = \bigcup_{i=1}^{m} \{\, \langle (v = i) \wedge c, l\rangle \mid \langle c, l\rangle \in \varrho(T_i) \,\}\,.$$

Clearly, for each assignment $x \in [m]^n$ there is a unique rule $\langle c, l\rangle \in \varrho(T)$ such that x satisfies c, and for this rule we have $f_T(x) = l$.

In the presence of classification noise each assignment on $[m]^n$ has the same probability η of being misclassified. Hence, if $R = \langle c, l\rangle$ is a rule induced by the target decision tree and the number of examples matching R is sufficiently large, with a high probability approximately a proportion η of these examples disagrees with R. This led Elomaa and Kivinen [1991] to define the notion of (κ, γ)-*accuracy* for $\kappa \in \mathbb{N}$ and $0 \le \gamma \le 1$, where κ gives a bound for the absolute and γ for the relative number of misclassifications made by a rule.

Definition For a rule R and a sample S, let $M(R, S)$ be the number of examples in S that match R, and let $D(R, S)$ be the number of examples in S that disagree with R.

The rule R is (κ, γ)-*accurate* with respect to S if $D(R, S) \le \max\{\, \kappa, \gamma M(R, S)\,\}$. A decision tree T is (κ, γ)-accurate with respect to S if all the rules induced by T are (κ, γ)-accurate with respect to S, and *strongly* (κ, γ)-accurate with respect to S if all the refinements of the rules induced by T are (κ, γ)-accurate with respect to S.

The property of having a strongly (κ, γ)-accurate decision tree is preserved when a sample S is split into the subsamples S_i^v, $i = 1, \ldots, m$, as demonstrated below. By T_i^v we denote the decision tree that is obtained from T by replacing each subtree T' that has the label v at the root with the i-subtree of T'. If v is the label of the root of T, the tree T_i^v is simply the i-subtree of T.

Lemma 18 *If T is strongly (κ, γ)-accurate with respect to S, then for all variables v and for all indices i, $i = 1, \ldots, m$, the tree T_i^v is strongly (κ, γ)-accurate with respect to S_i^v.*

Proof If a rule $R = \langle c, l\rangle$ is induced by T_1^v, then either R or the rule $\langle (v = 1) \wedge c, l\rangle$ is induced by T. In either case, if $R_1 = \langle c' \wedge c, l\rangle$ is a refinement of R, then $R_2 =$

$\langle (v = 1) \wedge c' \wedge c, l\rangle$ is a refinement of a rule induced by T. Hence, R_2 is (κ, γ)-accurate with respect to S. Since $M(R_2, S) = M(R_1, S_1^v)$ and $D(R_2, S) = D(R_1, S_1^v)$, the rule R_1 is (κ, γ)-accurate with respect to S_1^v. Thus, the tree T_1^v is strongly (κ, γ)-accurate with respect to S_1^v. In a similar way we see that the trees T_i^v, $i = 2, \ldots, m$, are strongly (κ, γ)-accurate with respect to S_i^v, $i = 2, \ldots, m$, respectively. □

Next we sketch a proof for the first part of the correctness of *Find*. That is, if there exists a decision tree T that is strongly (κ, γ)-accurate with respect to S and has rank at most r, then the call *Find*(S, r, κ, γ) returns a decision tree T' that is (κ, γ)-accurate with respect to S and has rank at most r.

Lemma 19 *If there exists a tree that is strongly (κ, γ)-accurate with respect to S and has rank at most r, then the call Find(S, r, κ, γ) returns a tree that is (κ, γ)-accurate with respect to S and has rank at most r.*

Proof It is clear from the definitions that *Find* never returns a tree that exceeds the rank bound r. It is also obvious that a tree T with the root labeled by v is (κ, γ)-accurate with respect to S if the subtrees of T are (κ, γ)-accurate with respect to the subsamples S_i^v, $i = 1, \ldots, m$, respectively. Hence, a simple induction shows that *Find* always returns either none or a (κ, γ)-accurate decision tree. It remains to show that the call cannot return none if the assumptions of the lemma are satisfied. This is done by a similar induction as in the proof of Lemma 9; we omit the details. □

We show next that it suffices to find (κ, γ)-accurate trees with suitable values of κ and γ. Recall, from the proof of Lemma 7, that by $L(n, r)$ we denote the maximum number of leaves in any reduced m-ary decision tree over $V_{n,m}$ of rank r. For the rest of this section, let T_* be a decision tree of rank r over n variables of arity m, and let P be a probability distribution on $[m]^n$. Let $\eta \leq \eta_b < 1/2$, $0 < \varepsilon \leq 1$, and $0 < \delta \leq 1$. Let S be a noisy sample of q examples of T_* drawn from $EX_\eta(P, f_{T_*})$, where

$$q \geq \frac{256L(n, r)}{\varepsilon^3(1 - 2\eta_b)^3}\left(mn \ln 2 + \ln \frac{2}{\delta}\right).$$

Finally, let

$$\kappa = \left\lfloor \frac{\varepsilon(1 - 2\eta_b)q}{8L(n, r)} \right\rfloor$$

and

$$\gamma = \eta_b + \frac{\varepsilon(1 - 2\eta_b)}{8}.$$

The following lemma shows that the function f_T represented by an (κ, γ)-accurate decision tree T of rank r fulfills the average disagreement condition required from a noise-tolerant Occam algorithm (Definition 5).

Lemma 20 *If a decision tree T has rank at most r and is (κ, γ)-accurate with respect to S, then*

$$\frac{D(f_T, S)}{q} \leq \eta_b + \frac{\varepsilon(1 - 2\eta_b)}{4}.$$

Proof Since T is (κ, γ)-accurate with respect to S, we have $\varrho(T) = A \cup B$, where $D(R, S) \leq \kappa$ for all $R \in A$ and $D(R, S) \leq \gamma M(R, S)$ for all $R \in B$. Since $|\varrho(T)| \leq L(n, r)$, we have

$$\sum_{R \in A} D(R, S) \leq \kappa L(n, r) \leq q \frac{\varepsilon(1 - 2\eta_b)}{8}$$

and

$$\sum_{R \in B} D(R, S) \leq \gamma \sum_{R \in B} M(R, S) \leq q\gamma = q \left(\eta_b + \frac{\varepsilon(1 - 2\eta_b)}{8} \right).$$

For all assignments x there exists the unique rule $\langle c, l \rangle$ in $\varrho(T)$ such that x satisfies c and $l = f_T(x)$. Therefore,

$$D(f_T, S) = \sum_{R \in \varrho(T)} D(R, S) = \sum_{R \in A} D(R, S) + \sum_{R \in B} D(R, S),$$

and the claim follows. □

It remains to show that with a high probability *Find* returns a (κ, γ)-accurate tree. Because of Lemma 19, we can show this by proving that the tree T_* is with a high probability strongly (κ, γ)-accurate with respect to S. In the proof we need the following result of probability theory by Hoeffding [1963]. Let $GE(p, t, r)$ be the probability of *at least* and $LE(p, t, r)$ the probability of *at most* rt successes in t independent trials each with probability p of success.

Lemma 21 *If* $0 \leq p, s \leq 1$, *and* t *is a positive integer, then*

$$GE(p, t, p + s) \leq e^{-2s^2 t} \text{ and}$$
$$LE(p, t, p - s) \leq e^{-2s^2 t}.$$

□

Lemma 22 *With probability at least* $1 - \delta/2$, *the tree* T_* *is strongly* (κ, γ)*-accurate with respect to* S.

Proof If $R = \langle c, l \rangle$ is a refinement of a rule induced by T_*, then $f_{T_*}(x) = l$ for all x that satisfy c. Hence, examples in S that disagree with R have been misclassified. If R is not (κ, γ)-accurate with respect to S, then a proportion greater than γ of the examples matching R have been misclassified, and there are more than κ such examples. Let $s = \gamma - \eta_b$ and $u = M(R, S)$. Lemma 21 gives an upper bound

$$\begin{aligned} GE(\eta, u, \gamma) &\leq GE(\eta, u, \eta + s) \\ &\leq e^{-2s^2 u} \\ &\leq e^{-2s^2(\kappa+1)} \\ &< e^{-2s^3 q/L(n,r)} \\ &\leq e^{-mn \ln 2 - \ln(2/\delta)} \\ &= \delta/2^{mn+1} \end{aligned}$$

for the probability that the rule R is not (κ, γ)-accurate with respect to S. The number of refinements of rules induced by T_* cannot exceed the number of conjunctions over single value assertions on variables $v_1, \ldots, v_n$. The number of such assertions is certainly bounded by 2^{mn}, since there are m single value assertions on each of the n variables and each of them can be either present in or missing from the conjunction. By summing these probabilities over all the rules R we therefore see that with probability at least $1 - \delta/2$ all the refinements of the rules induced by T_* are (κ, γ)-accurate with respect to S. □

We are now ready to prove that *Rodt* is a noise-tolerant Occam algorithm:

Theorem 23 *Rodt is a noise-tolerant Occam algorithm for decision tree of rank r.*

Proof The run time of *Find* is $O(qm^r(n+1)^{2r})$; the analysis is almost identical to that carried out in Lemma 10.

By Lemma 22, with probability $1 - \delta/2$ there is at least one decision tree of rank r that is strongly (κ, γ)-accurate with respect to S, namely T_*. By Lemma 19, *Find* returns a (κ, γ)-accurate tree of rank at most r, and by Lemma 20 this makes *Rodt* a noise-tolerant Occam algorithm. □

Corollary 24 *Decision trees of rank r are polynomially learnable in the presence of classification noise.*

Proof Immediate from Theorems 6 and 23, since by Lemma 7 the class of decision trees of bounded rank is of polynomial size. □

In noise-free domains the iterative algorithm *Findmin* learns decision trees of fixed rank without receiving an explicit rank bound as input. The same technique can be employed in the noisy setting: the correctness of *Rodt* guarantees that, with probability $1 - \delta/2$, the first returned tree is a (κ, γ)-accurate decision tree such that its rank does not exceed the rank of the target tree (assuming the sample size is large enough).

3.5 The algorithm Rank

In this section we gather the preceding refinements and improvements into a decision tree learning algorithm that combines ideas from theoretical studies and practical experience. We refer to the algorithm as *Rank*.

A practical learning algorithm has to be efficient in order to be generally applicable. Therefore, we, for instance, do not have time to iterate the learning process in order to find a good approximation of the noise rate affecting the learning situation as proposed in the previous section. Instead, we expect the user to supply appropriate values for κ and γ. It is typical in inductive learning methods to expect the user to supply a value for a confidence level or a threshold parameter ("γ"). For instance, ID3 [Quinlan 1986b], CN2 [Clark & Niblett 1989], and C4.5 [Quinlan 1993] are all examples of such programs. Furthermore, in C4.5 the user is allowed to tune the value of a parameter that corresponds to κ. Resorting to the user's choice of parameter values, naturally, loses the general provability of the method, but still, if appropriate values, as given in the preceding theorems, are supplied, the guaranteed learnability properties of the following method are preserved.

We have to deviate from the formal considerations, also, in what concerns the sample size. The algorithm is expected to produce a sensible hypothesis even with the smallest samples. Therefore, we apply the following heuristics in the learning algorithm. If no tree is found, then we resort to predicting the most common class; i.e., we return a single leaf tree that is labeled with the most common class. The situation may arise, for instance, if strict match is required for an inconsistent sample.

The main program of our method is presented as Algorithm 3.7. We have given default values to the input parameters of the algorithm; they do not fit all learning domains. The given values suit perfect domains, where heavy pruning is not needed.

The following small trick in *Rank*'s high level control structure makes the algorithm in practice much quicker than *Findmin*. There is no need to seek to examine the rank candidates in order starting from value 0. The search can be started from any candidate value (in between 0 and n).[2] The program divides into two separate parts: In one part (Lines 2–4) the case, where no tree was recovered using the initial rank candidate, is handled and the other part (Lines 5–8) takes care of the situation where a tree was returned. The former part entails increasing the rank candidate until a decision tree is returned by *Find*. Even though the final part of the algorithm works in the same manner as in *Findmin*, something has been gained by this little trick; viz., rank candidates smaller than the initial value, which would not have been successful, have been passed over without unnecessary inspection. In the second part, where a tree was already recovered with the

[2] If the typical rank values had a wide range, *binary search* could be utilized here. However, real-world domains tend to have a relatively low rank (see Chapter 5). Therefore, it is not worth the effort to incorporate a sophisticated search technique here.

Algorithm 3.7 *Rank*$(S, R = 2, \kappa = 1, \gamma = 0.95)$
input: a nonempty (possibly noisy) sample S of some function $f : R_1 \times R_2 \times \ldots \times R_n \to R_C$.
The algorithm may optionally be supplied with the following additional parameters:
Nonnegative integers R and κ, and a real γ, such that $0 \leq \gamma \leq 1$.
output: a decision tree.
begin
(1) $r \leftarrow R; T \leftarrow$ *Find*$(S, R, V_{n,m})$;
(2) **if** $T = \text{none}$ **then**
(3) **repeat** $r \leftarrow r + 1; T \leftarrow$ *Find*$(S, r, V_{n,m})$ **until** $T \neq \text{none}$ **or** $r = n$;
% let k be the most common label among the examples in S.
(4) **if** $T = \text{none}$ **then** $T \leftarrow T = k$ **fi**
else
(5) $r \leftarrow r(T)$;
(6) **while** $r > 0$ **do**
(7) $Q \leftarrow$ *Find*$(S, r - 1, V_{n,m})$;
(8) **if** $Q \neq \text{none}$ **then** $r \leftarrow r(Q); T \leftarrow Q$ **fi**
od
fi;
(9) **return** T
end.

initial rank candidate, it remains to check whether it is of minimum rank. Again, we do not have to run *Find* for all consecutive rank candidates, but it suffices to check the values that are smaller by one than the smallest rank of a decision tree found thusfar.

The only remaining major modification to the control of *Findmin* in *Rank* is the most common class prediction in case that no classifier with required properties is found (Line 4). Also, upward search can be terminated if no hypothesis is found before the rank candidate exceeds the number of attributes (Line 3). No classifier on n attributes can have rank beyond the limit n. This stopping condition was not needed in the earlier *Findmin* variants, since a consistent sample always has a classifier with rank less or equal to the number of attributes in the domain.

This method's time requirement is only linear in the number of training examples and thus it fulfills even the strictest efficiency requirements posed to practical learning programs [Clark & Niblett 1989].

One practical aspect of this program that has not been made explicit is its missing value management. We simply apply Clark and Niblett's [1989] technique of filling in a missing value with the most common value of that attribute. We choose this method because it is quite simple and relatively competitive [Quinlan 1989] and, hence, suffices to us in this study, where we do not emphasize this aspect of learning.

The incremental method is not incorporated into the above described algorithm. The same control optimizations that can be used in batch learning are not applicable in incremental learning. Therefore, it is not possible to intertwine *IFM* totally to *Rank*. Instead, we run *IFM* separately whenever incremental learning is needed. Of course, the algorithms have been implemented together, and the same subprograms are utilized by both high-level control procedures. It should be clear that the simulation scheme of *IFM* can be applied in the noisy setting as well. In that case we just call the version of *Find*, which provides for noise, from *IFM* as well.

Chapter 4

TELA—a Tool for Attribute-Based Induction

Testing and comparing different implementations and techniques on sample data is an integral part of the design and application of inductive machine learning programs. It involves many simple but tedious auxiliary tasks that are usually not supported by the learning tools themselves. Hence, there is a strong need for an environment that facilitates testing and comparing different inductive learning programs. Most of the facilities that are needed in such an environment have to do with data set manipulation.

This chapter describes TELA (Testing Environment for Learning Algorithms), an integrated environment that has been developed to alleviate the troubles caused by the unavoidable subsidiary tasks when experimenting with inductive learning programs. It is distributed freely to all interested parties.[3] TELA incorporates facilities and support for data transformation, format conversion, experiment design and execution, and statistics collection. It has been designed to accommodate, in principle, any program using an attribute-based representation formalism. The initial development of TELA is described by Becks *et al*. [1992] and Elomaa *et al*. [1995]. We present the design rationale of the system, describe its current state, and consider future enhancement of it: In Section 4.1 we concentrate on the design principles of TELA. Section 4.2 contains a detailed description of the current system. Future development directions of TELA are outlined in Section 4.3. Related systems are surveyed in Section 4.4. Finally, Section 4.5 reviews some practical experiences gained about TELA.

[3] TELA system together with a comprehensive documentation is available from the URL-address `http://www.cs.helsinki.fi/research/pmdm/ml/tela.html`. Alternatively, it can be obtained by anonymous `ftp` access from `ftp.cs.helsinki.fi/pub/Software/Local/TELA`.

4.1 Design rationale of TELA

Consider the following situation: Data has been acquired from the application domain, it has been classified for use as a training set for a learning algorithm, and the data has been prepared into the format required by the algorithm. Instead of simply leaving the assessment of an acquired classifier to the domain expert, we can assist the process by testing and evaluating the rule in several ways. We might assist the assessment by empirically testing the classifier by repeatedly dividing the training set into (random) training and test sets, we might also experiment with modified versions of the data, e.g., leaving some attributes out, or we might try different induction tools on the same task. All the data transformations and format conversions needed in the tasks mentioned above can surely be accomplished with the help of standard `UNIX`, say, tools and a text editor. However, these secondary subtasks require much more effort than the primary objective of experimenting with the algorithms and data sets and, hence, easily drown the sight of what really is important.

Since experimentation is an integral part of the design process of new learning algorithms [Langley 1988] as well as application of inductive programs, as exemplified above, it is essential that experiments can be carried out without unnecessary complications. TELA is a system that has been designed to facilitate experimentation with attribute-based inductive learning programs. In the following we motivate the design of the system.

There are two general goals that have been pursued in the development of TELA—*independence* and *uniformity*. Both goals have manifold aims and manifestations in the system.

- Independence of particular learning algorithms, classifier representations, and visualization tools produces an environment that is as general and versatile as possible. Commitment to particular algorithms or classifier types (e.g., to TDIDT learners) always rules out related algorithms or relevant approaches (e.g., decision list learners). Binding the system's interface to a particular visualization tool mainly hampers its portability, but may also bias against new types of learning algorithms.

- Uniformity of the appearance of the system and its facilities gives the user better control over the system and lets him, thus, concentrate on his main duties: Experimentation with and comparison of learning algorithms. Controlling a too diverse environment is an unnecessary and interfering task.

In addition to the general goals, specific operational objectives have been put to TELA. Today a number of different inductive algorithms are easily available (by anonymous `ftp`, for example). Even though many algorithm implementations come with the source code, one would like to retain the black-box-view of them, but still be able to compare them with other algorithms or implementations. In other words, it should be enough to familiarize oneself only with the input/output of a given algorithm before experimenting with it. Hence, a system is needed that can easily accommodate (almost) any learning algorithm that uses attribute data. Such a platform should support the following activities, which are all featured in TELA.

Data preparation and manipulation

- Extracting basic statistical information of the data sets involved. In addition to data carefully designed by the domain expert, the learning algorithms typically have to deal with data acquired from some sensors or a database. In order to be able to reliably assess the performance of an algorithm, the user has to be aware of the basic statistical facts (representativeness) of the training and test data.

- Easy declarative definition of the data formats used by the learning algorithms and automatic conversions between these data formats are required to lighten the addition of new learning algorithms and for executing unbiased comparison of different algorithms on exactly the same data.

- Dynamic data manipulation by changing the sets of attributes and examples considered and the possibility to transform data by redefining attribute types and by adding random noise to the examples. This allows the user to vary the application domain characteristics as he pleases. For example, noise-tolerance may be of prime importance when the learning algorithm is put to actual use.

Execution control

- Efficient communication between the environment and an enclosed learning algorithm. Even though the platform and learning algorithms are independent of each other, the user has to be able to operate the algorithms from within the environment and receive feedback from them.

- Experiment planning and incremental execution of tests are central vehicles of a successful environment. Without adequate support for higher level experiment compilation the operation of such an environment reduces merely to run time support for learning algorithms. Incremental execution of tests is a desirable property because experimentation, by nature, requires re-evaluating (and rerunning) tests once (partial) results are reflected upon.

- Support for multiple users is required in any program that endeavors to be a general-purpose tool. Control and monitoring of users must ensure that different users will not get tangled into each others experiments.

Analysis support

- Standard ways of testing the algorithms and standard ways of relating their performance by measuring suitable execution statistics. Commonly accepted test strategies (e.g., repeated random partitioning and cross-validation) should be available for all algorithms. Furthermore, the same performance measures (e.g., average accuracy, size, and learning time) should be measurable for the classifiers of all algorithms.

- Benchmark methods for validating the results produced by the learning algorithms. The user can achieve an understanding of the general usefulness of the machine learning algorithms on an application by relating their performance to that of a simple well-established method (e.g., Bayes-rule [Breiman *et al*. 1984] or one-level decision trees [Holte 1993]).

In the next section we describe in detail how TELA meets all these demands. In addition to the visible functionality listed above, the platform needs a number of internal supporting functions. The overall architecture should offer the user a coherent view of the many diverse functions that are available. Coherence is achieved in TELA by including most of the above functions into a single command language.

One of the larger issues that we had to resolve when designing TELA was to decide on the extent of the system's internal knowledge representation language (KRL). A common KRL is required to communicate data between algorithms. It cannot simply be left to the user to define the translation from the data format of the new algorithm to those of all previously incorporated ones, since there may be tens of such algorithms already present. Hence, an internal representation language, acting as an intermediator in all these translations, is required. Even though we are operating within the restricted world of attribute-based representation languages, there is plenty of room for variation. One could aim at developing a universal KRL for learning algorithms that use the attribute representation. Such a language ought to be able to express all inputs, outputs, background knowledge, and, even, intermediate results of all algorithms within its scope. But that makes the representation language extremely vulnerable: Any change in the data representation of a single algorithm requires changing the KRL and propagating the changes to all other algorithms. Instead of providing a universal KRL, TELA implements the common core of the data representation languages of attribute-based induction algorithms. The same arguments have guided the KRL design in the *Machine Learning Toolbox* project [Causse *et al*. 1990, Sleeman 1994].

Communicating the induced classifiers between two algorithms makes sense seldom—for example, a neural network cannot usually be interpreted as a decision tree. Moreover, management of the multitude of output formats of inductive algorithms would lead to close intertwining of the system and the algorithms. Hence, we have not tried to represent the output classifiers in the internal KRL of TELA. It is left to each individual learning algorithm to represent and interpret its own classifiers.

4.2 An overview of the system

This section describes TELA at its current composition (version 1.995). We demonstrate how TELA implements the requirements stated in the previous section. First we describe the general architecture of the system. Next we introduce the high-level experiment specification language that is the main distinction between TELA and its closest equals; detailed discussion is left to Section 4.4. Finally, we describe the facilities for semiautomatic accommodation of new learning algorithms. More detailed description of the technical aspects can be found in the user manual [Elomaa & Rousu 1996a].

4.2.1 System architecture

A general view of the architecture of TELA is presented in Figure 4.1. The user interacts with the system chiefly in terms of the language TESLA (TELA Experiment Specification Language). The language contains facilities for accomplishing the operations demanded in the previous section. Only seldom—when enclosing new algorithms to the system—is there need to use the other language of TELA, the data format definition language (Section 4.2.3). Hence, the appearance of the system to its user is coherent.

The three modules of TELA that are visible to the user are Data Logging Module, Experiment Specification Module, and Data Format Definition Module. A brief description of these components follows.

Data Logging Module assists the user in defining the data. Most learning algorithms require the user to supply attribute declarations in addition to the actual example vectors. Now, if a large database of examples is received from the application domain without attribute declarations, a skeleton attribute declaration file can be generated with this module for any learning algorithm, whose data format has been defined, instead of going through the voluminous data and manually recording the values appearing in the data for each of the attributes. The skeleton file lists the attributes (with dummy names) and the values (or the subrange of values) that appear in the example vectors for that attribute. The domain expert can then give correct names for the attributes and extend their ranges, if required. If the domain expert carefully designs the example set, not much assistance is usually needed. Moreover, the example volumes tend to remain low in these cases.

Experiment Specification Module is the main method of interaction between the system and its user. This module contains the language TESLA that is used to specify the test sequence. Execution of TESLA specifications is interactive and incremental. This is accomplished with the SEED session editor interface [Holsti 1989]. At any given moment, usually as a consequence of a response given by the system, the user can go back in

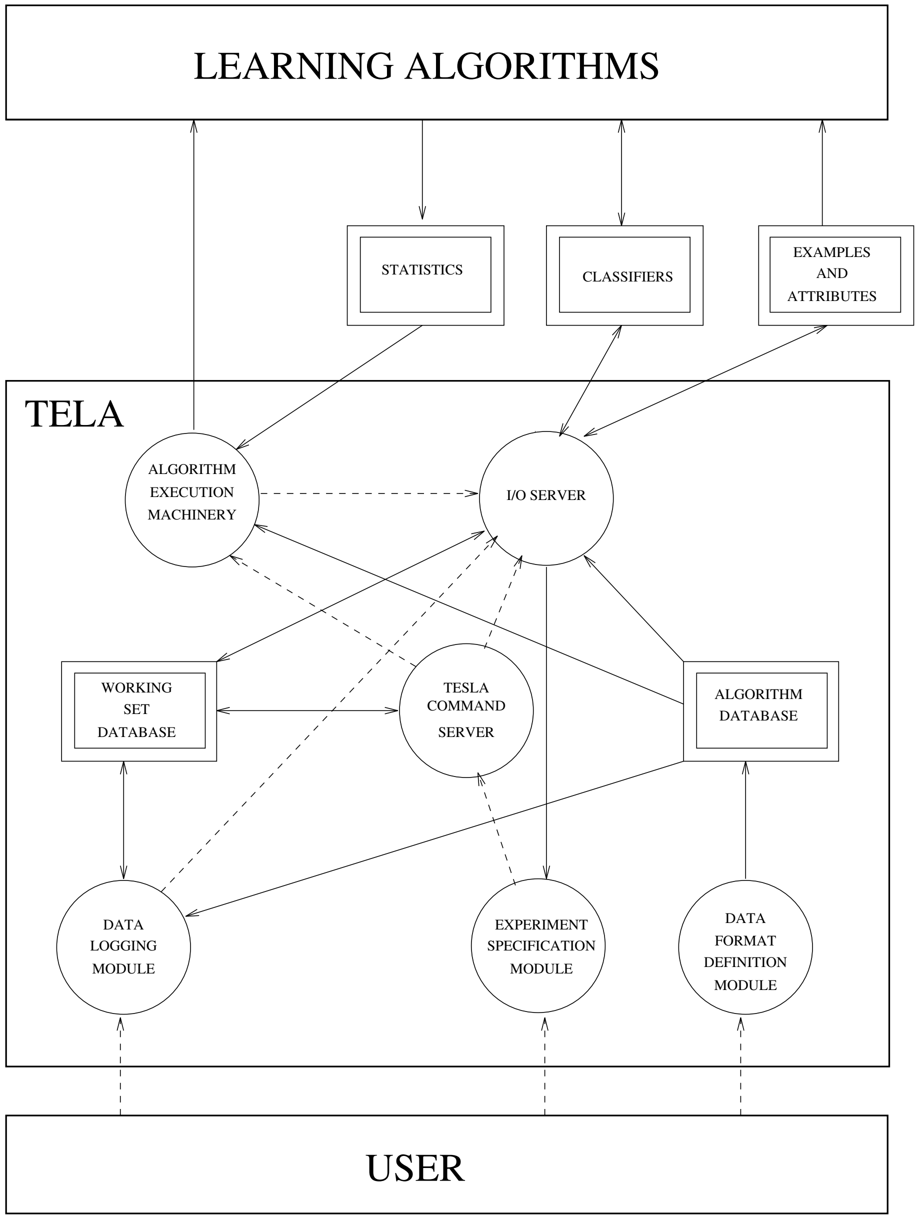


Figure 4.1: The general architecture of TELA.

the TESLA code and change the specification given; parts of the specification are automatically retracted and re-executed to update the state of the system to correspond to the changed specification. The final specification can be saved as a record of the experiment.

Data Format Definition Module is invoked when a new learning algorithm is added to TELA. This module automates the generation of parsers and scanners for the attribute declarations and example descriptions of the new machine learning program. It also introduces new terms to the language TESLA. The user only has to provide a simple description (fill in a template provided) of the syntax used in the attribute declarations and example descriptions. Algorithm parameters are also introduced in this description for TESLA; however, they are not processed automatically, but need the user to provide an external processor to interpret them in terms of the learning algorithm. Similarly, management of the output of the algorithm has to be done manually. This includes classification, display, and accuracy recording.

The internal modules of TELA which are not directly manipulated by the user are the TESLA Command Server, two Databases, the Algorithm Execution Machinery, and the I/O Server.

- *The* TESLA *Command Server* transforms the specification provided by the user into calls of the internal functions and services of other modules.

- *The Algorithm Database* records and maintains information about the learning algorithms. The data format definitions for learning algorithms are maintained in the database; they are queried by the other internal modules.

- *The Working Set Database* contains information about the active attributes and examples, and the active classifier, if any.

- *The algorithm Execution Machinery* calls external learning algorithms and controls their execution in cooperation with the I/O Server. It is responsible for gathering the requested statistics from the external algorithms. Algorithm dependent particularities are queried from the Algorithm Database.

- *The I/O Server* passes information between the system and the learning algorithms; it takes care of the data conversions that are needed in the process. The attribute declarations and example vectors are communicated through files.

There are two standard benchmarks in the system. The results obtained by a learning algorithm can always be contrasted with the results that would be obtained using any of the *Rank* variants or Rousu's [1996] MDLP-based learners. No other algorithms are distributed with the system (because the proprietorship of the algorithms belongs to their developers). However, the I/O interfaces for the algorithms NewID [Boswell 1990b], CN2 version 4.1 [Boswell 1990a, Clark & Boswell 1991], C4.5 [Quinlan 1993], and ITI [Utgoff 1994], Naive Bayes [Kononenko 1993], and T2 [Auer *et al*. 1995] are part of the current system.

4.2.2 Experiment specification language

A central feature of TELA is its high-level procedural experiment specification language TESLA that lets the user define complicated test sequences and comparisons with the learning algorithms and data he chooses. The language allows recording intermediate results and saving classifiers for further inspection. Incremental execution and manipulation of TESLA specifications is supported by the SEED session editor interface [Holsti 1989].

TESLA is a procedural language that contains statements for accomplishing most of the effects listed in Section 4.1. There is only one control statement in TESLA: Repeated execution of test sequences is enabled by the do statement. The other statements of TESLA are simple statements that do not change the control of execution. A typical TESLA specification contains operations from each of the three categories mentioned in Section 4.1. The statements of TESLA can be assigned to these categories as follows.

- Data preparation and manipulation operations dominate the language. There are eleven statements for these purposes: read, write, stat, exclude, include, select, map, class, randomize, noise, and divide.
- The only execution controlling facility in TESLA is the run statement.
- Three statements for (directly) supporting classifier analysis are included in TESLA: test and cv allow empirical testing of a classifier; do implements iteration. Some of the aforementioned statements also support empirical analysis, but less directly.

In Table 4.1 an example TESLA specification is given, in which, first, a set of active attributes is declared and the active example set is defined with the read statement. The attribute declarations and the example vectors must conform to the predefined data formats, here they follow that of the C4.5 algorithm [Quinlan 1993]. A classifier saved in a file could also be activated with this statement. Algorithm dependent default file name extensions are automatically expanded to the file names given.

After loading the data, the range of attribute "date" is dynamically redefined using the map statement. Using this command a set of values or a subrange can be mapped to a new value; the redefined attribute will always be of nominal type. Note that we implement implicit value hierarchies using this statement by mapping an existing attribute's values (the months) to those on a higher level of abstraction (seasons of the year). After changes the new distribution of the attribute's values is queried with the stat statement

Command select is used to choose from the active set of examples those ones that have "summer" as the value of attribute "date". This renders the attribute superfluous and, hence, it is excluded from the experiment.

Two irrelevant attributes are dynamically declared using the include statement, which replicates each example in the data as many times as the new attribute has possible values. Hence, the number of examples becomes multiplied by four in our example spec-

Table 4.1: Example of a TESLA specification. In this actual interaction user commands are marked with bold font. TELA's replies have been, in part, abbreviated for better legibility.

```
Welcome to TELA version 1.995 !
```
read C45 atts soy exs soy;
```
Attribute description read from file soy.names.
290 examples read from file soy.data.
```
map date as [april may]: spring [june july august]: summer else fall others winter;
```
Mapping complete.
```
stat date;
```
 Attribute          Type   Values       ...  Unknown  Don't Care
 name (* = class)          (Nom.)             count      count
 date              DISCR                        0          0
                           fall    101
                           spring   43
                           summer  146
                           winter    0
Total example count:  290
```
select date in [summer];
```
146 example(s) selected.
```
exclude date;
```
Specified attributes excluded.
```
include irrel1: a b, irrel2: c d;
```
Working set contains now 37 attributes and 584 examples.
```
set randseed 0;
```
Random number generator initialized with seed 0.
```
randomize class: 0 others 5;
```
class:  0 out of 584 values changed.
areadama:  24 out of 584 values changed.
cankerle:  19 out of 584 values changed.
cankers:  26 out of 584 values changed.
...
```
do 10 record time size rank accuracy;
   divide training: 67 others testing;
   run Rank (initrank=2, gini, level=4, kappa=1, gamma=0.905);
   read TELA both testing;
   test Rank;
end;
```
RUN Rank rounds:  10 AVG.RANK =2.000 (+/-0.000) AVG.SIZE =134.300
                     (+/-6.870) AVG.TIME =10.034 (+/-0.403)
TEST Rank rounds:  10 AVG.ACCURACY =81.255 (+/-0.287)
```

ification. Assigning each of the values to a replicated example ensures that the new attribute will not be relevant to the class label of the examples.

Before the random number generator is applied we initialize its seed to a constant value as to obtain a test sequence that can better be repeated. Random noise rate of 5% is added to all attributes' values using randomize statement. It allots, with probability 0.05, a new value for an attribute in any example vector. All values in a (discrete) range have an equal probability of being chosen. In particular, the command may also leave an attribute's value intact. The other randomization command of TESLA, noise, does not include the attribute's original value in the set from which a new value is allotted.

Any legal sequence of legal TESLA statements can be repeated any number of times using the do statement. The repetitions are independent; i.e., the starting configuration of the system's internal state is the same for each repetition. Nested repetitions are not allowed. In this example the do statement records average values over 10 repetitions for the construction time, size, accuracy, and rank of the resulting decision tree.

Within the do statement the active set of examples is split, using the divide statement, into two mutually exclusive portions, containing 67% and 33% of the data. The first portion remains active. All designated portions are written (in TELA format) into files from where they can later be reactivated.

After desired manipulations and transformations have been performed on the data, the *Rank* algorithm is run on the modified data. The run statement invokes *Rank* with the given parameter values. The active sets of attributes and examples are communicated to the algorithm. The returned classifier will become the active one. After activating the test data, the classifier is tried using the test statement, which simply checks the accuracy of the active classifier on the set of active examples.

The remaining statements, which were not included in the above example, are as follows. At any time, the user can save (modified) attribute declarations, examples, or the active classifier into a file by the write statement. The statement does not change the activation situation of the system: All sets that are active when the statement is invoked will remain active after its execution. New class attribute can be chosen using the class statement. The new class, naturally, has to be of nominal type.

Cross-validation testing [Breiman *et al*. 1984] can be accomplished in TESLA by the cv statement. In cross-validation the data is split into n disjoint subsets of (nearly) equal size, then, for each subset a classifier is constructed leaving the one subset out of the training set and using it as the test set. The average of the n independent error estimates is a very good approximator of the true misclassification rate of the classifier.

4.2.3 Enclosing new algorithms in TELA

The first duty of TELA is easy accommodation of new attribute-based learning algorithms without having to know the particularities of them apart from their I/O formats.

Table 4.2: Data format definition for the *Rank* algorithm.

```
BEGIN DESCRIPTION  Rank        % Inherits unspecified values from
        DEFAULTS = TELA        % the description of TELA
        RULETYPE = 'Decision Tree'
[ATTRIBUTES]
        ATTSEP = ';'           % Separator of consecutive defs
        VALSEP = ','           % Terminates a value's name
        ATTSEXSSEP =
        NOMINAL =
        INT =
        CONT = 'CONT'          % Required by TELA
        UNKNOWN = 'x'          % Symbol for unknown value
        DONTCARE =
        ATTSFOOTER = 'end'     % List terminator
        EXSFOOTER = 'end'
[INTERFACE]
        EXENAME = 'algorithms/rank/rank.run'  % Parameter processor
        ATTSEXSFILES = SEPARATE               % (see Appendix C)
        ATTSEXT = '.att'       % Default extensions
        EXSEXT = '.exs'
[OPTIONS]
        GINI                   % Legal parameters
        IFM
        PREVHYPO
        INITRANK
        LEVEL
        KAPPA
        GAMMA
[RUNSTATS]
        SIZE                   % Quality measures
        RANK                   % TIME is present for all algorithms
[TESTSTATS]
        ACCURACY
END DESCRIPTION
```

For that purpose TELA offers a declarative data format definition language. The input to a (non-incremental) learning algorithm can be divided into two parts: *Static* input consists of the attribute declarations and of the data vectors conforming to the declarations; the *dynamic* part of the input of an algorithm consists of its commands or parameters and the values subscribed to them. The definition language of TELA follows this division: Definitions have a static and a dynamic part. However, not all data formats have a dynamic part: The internal knowledge representation language of TELA can only be used in describing the attributes and the examples, but it has no executable correspondence.

TELA provides the user with a template to be filled in for the data format definition. This template has slots for a few simple declarations (e.g., different separators used in the input files) defining the specifics of the new algorithm's attribute declaration and example description syntax. Default values to this definition are inherited from the TELA data format, but the user is free to any other existing data format as the superclass of the new definition. Hence, only those slots that differ from a predefined data format need to be filled in. In Table 4.2 *Rank*'s data format definition is given. The declarations are stored into the Algorithm Database from which they subsequently queried by the other internal modules of TELA. Predefined standard declarations belonging to the current version of TELA are TELA (the internal KRL, static only), NewID, CN2, C4.5, MDLTree, MDL-List, Rank, Findmin, ITI, T2, and Naive Bayes. The dynamic control cannot easily be automated in TELA. Therefore, it expects to receive the name of an executable external program that can process and pass, e.g., parameter values to the learning algorithm (cf. Table 4.2). The external C shell program of *Rank* is presented in Appendix C.

As already explained, the outputs of different learning algorithms cannot be handled as smoothly. The user must provide TELA with procedures for saving and visualizing the produced classifier, for classifying a set of examples with it, and for recording the accuracy of a classifier on a given example set. In the simplest case (e.g., for NewID), the user can implement these with the help of the learning algorithm itself: The classifier can be written into a file and can be offered for user inspection (in ASCII format) with a command of the learning algorithm, subsequently the classifier can be reread by the algorithm and used to classify a set of examples. Only recording the accuracy of a classifier on a given data causes some troubles, but typically that information is easily extracted from the output of the learning algorithm. In a more difficult case (e.g., for C4.5), the algorithm neither supports saving the classifier in user-readable format nor allows classifying a set of (unseen) examples with it. In such a case the user has to implement these features (e.g., in C code).

4.3 Future development of TELA

Even though TELA has been implemented and used to execute the experiments reported in the next chapter and in experiments of other studies [Lamminjoki 1995, Rousu 1996, Elomaa & Rousu 1996b], it is still a subject to future improvements. This section briefly takes up the most urgent development needs of TELA.

At its present composition TELA lacks a graphical user interface, which might be useful in some situations. The main reason for this omission is that we have wanted to be able to distribute the system with few environment restrictions. Furthermore, our experience of decision tree learning has demonstrated that in practice the induced trees often are of such magnitude that there is no good way of visualizing them (and decision lists do not really need graphical representation). However, extending TELA with a graphical user interface has been taken into account in the system's design; it should be quite easy to extend TELA with an independent graphical user interface.

Even more than graphical extensions TELA needs support for the TESLA language, which is a sizable programming language. A programming environment or, at least, a syntax editor would facilitate test sequence specification.

Currently TELA can only support a single set of attributes and examples and a single classifier. Together these three components are called a *working set* [Tsatsarakis & Sleeman 1993]. If the user wants to manipulate several working sets in TELA, he has to manage them (save and load, as required) himself. It is quite typical that slightly different working sets are manipulated during a test sequence. Therefore, automatic control over working sets should be added to TELA. Implementing such a property in TELA is straightforward: The language TESLA just needs tools for naming and accessing different working sets. The Working Set Database can maintain the information regarding the working sets.

In inductive learning tasks it often is the case that the attributes supplied are not definitely the ones that should be used. If, however, we have a fixed set of data from the application domain at our disposal, we cannot expect to find new information about the data, but we have to content ourselves with that at our disposal. Nevertheless, in many cases reorganizing the fixed set of data by constructing new attributes from the existing ones can turn out beneficial either in classification accuracy or intelligibility of the resulting classifier. There are learning algorithms that automatically construct new attributes from the primitive ones [Matheus & Rendell 1989, Pagallo & Haussler 1990], but that is not the case for all learning algorithms. A facility that would give the domain expert some tools for constructing new attributes from the existing ones, e.g., as algebraic combinations of numerical attributes, could be added to the system. At the moment the map statement of TESLA is the only tool in TELA for doing anything of this kind.

More sophisticated measurements of data characteristics [Michie *et al.* 1994] could be presented to the user for him to better be able to relate his results and assess the utility of an algorithm. In principle, nothing prevents including such into the system. Taking the idea even further, tools for automatic parameter value adaptation or even learning method selection (multistrategy learning) would suit TELA's framework well. However, these would extend the system far beyond its original target functionality.

There are many further extensions that could be considered to TELA (e.g., automatic comparison of working sets) and extensive use will surely bring out many more. Instead of trying to list all of them, let us just note that the strict discipline exercised in TELA concerning the independence of the system turns out to be valuable in this respect. It is easy to extend TELA with new facilities. Of course, some desirable properties are fundamentally hard to implement; they will not be any easier in connection of TELA.

4.4 Related systems

There are other systems for supporting the execution of inductive learning algorithms. WILA [Sleeman 1994, Tsatsarakis & Sleeman 1993] is closest to our system in its design rationale and functionality, even though its aims are in part different from those of TELA. WILA is a prototype system for helping both the domain expert and the knowledge engineer to use attribute-based inductive algorithms. WILA shares our platform approach in that it, too, is intended to be an easily extensible environment for independent learning algorithms (as published, WILA only incorporates the NewID algorithm). Facilities for preprocessing the data to be fed in are provided for the domain expert and the knowledge engineer. They can also postprocess the output of the learning algorithm.

WILA differs from our system most dramatically in that it offers only minimal support for experiment planning, execution, and analysis. No special facilities for supporting test sequence design or execution are provided. WILA only gives the user the opportunity to do some pre- and post-processing for single runs of NewID. TELA offers many facilities for relating the performance of different algorithms on the same task or a single algorithm on different tasks, but WILA mainly relies on the graphical presentation of the induced decision tree and the domain expert's understanding of it. Furthermore, the view offered to the user is quite dispersed in WILA: A separate tool is provided for each task. Finally, WILA lacks many of the data manipulation opportunities that are part of our system. On the other hand, WILA is able to manage multiple working sets, it has a limited facility for building new attributes from existing ones, and it has a graphical user interface.

IND [Buntine & Caruana 1993] is a comprehensive system implementing the most popular approaches to decision tree learning as variations of a common learning algorithm. For example, IND implements C4.5 by executing the appropriate splitting rule selection and pruning methods when constructing a decision tree. IND has many advanced features and it is quite a versatile tool for decision tree induction. Its approach, however, is fundamentally different from that of TELA and WILA. First, IND is a closed system that is not intended for the user to extend. If a new decision tree learning method, nevertheless, was to be added to IND, then, in the simple case, one could add a new style to IND's repertoire by running precoded programs with suitable options. In the more difficult scenario, the new algorithm cannot be implemented directly by IND, but requires adding new programs or options to the system. In both cases the user has to have in-depth knowledge of the new method. Furthermore, implementing decision list learning algorithms—CN2, for instance—is not supported by IND, even though the algorithms principally use the same techniques as decision tree learners.

GENCOL of De Raedt and Bruynooghe [1992] is a very general implementation of a concept learning framework. Most approaches to concept learning can be implemented in GENCOL by instantiating its many generic parts. The generality of the system makes it more a tool for the design of new concept learning approaches than a tool for practical induction tasks.

4.5 Practical experiences on using TELA

This section reflects lessons learned using TELA. Both personal experiences and those of other users are offered. Up to these days TELA has been used solely by the members of the development group. It is only quite recently that TELA has been adopted into use by other teams. It is time to review the first feedback on TELA's design and utility.

Inclusion of new learning algorithms

Several learning algorithms have been incorporated with TELA since the completion of the system. For this work algorithms *Rank* and ITI [Utgoff 1994] have been added, Rousu [1996] has incorporated two MDLP-based algorithms, and the T2 algorithm [Auer *et al*. 1995] has been made part of the system's repertoire.

The current implementation of *Rank* was modified, especially with TELA in mind, from an older version of the algorithm [Elomaa 1992]. Therefore, it was the easiest task to incorporate *Rank* into TELA; all the potential pitfalls could be avoided by taking them into account already in the implementation. As an example, the algorithm counts and outputs (in the desired format) all the required statistics, and is able to input and output its decision trees. Hence, inclusion of *Rank* into TELA only required us to give the data format definition (Table 4.2) and to modify a suitable dynamic control program for *Rank* (Appendix C) from that of C4.5. The latter duty was clearly the more demanding one, and even it could be handled quite smoothly.

Lamminjoki [1995] enclosed the incremental decision tree learner ITI [Utgoff 1994] into TELA's repertoire. He was not familiar with the algorithm or the environment in advance, but had ample knowledge on inductive learning. According to Lamminjoki's experience it was relatively easy to handle the dynamic control of ITI. Size of the resulting tree was easily extracted from among the output of the algorithm, the decision tree could be saved into a file for later use, classifying a set of test examples with a tree grown was readily available in ITI, and its accuracy, again, was easy to extract. What turned out to be more complicated, was the measurement of the rank of a tree produced by ITI. Naturally, it was not featured in ITI itself, but because of the experiments in the next chapter we, however, wanted that measure at our disposal. In order to save time and effort, it was elected to use a shortcut in obtaining this measurement: Rather than implement a parser and a scanner for ITI's decision trees (which has been done, e.g., for C4.5) we simply added a rank-counting function to the source code of ITI. In general, it seems to be the biggest shortcoming of the current version of TELA, that often one would need to manipulate the hypothesis generated by an algorithm, but no support for that is provided. In summary, it is Lamminjoki's view that it is not hard to embed an unfamiliar algorithm into TELA (as long as it is not too peculiar) as regards the basic functionality and measurements; all additional requests take much more effort.

The experiences of Rousu [1996] on enhancing the repertoire of TELA are parallel with those of the author: It is easy to make ones own algorithms to conform to the requirements of the system. Similarly, the problems encountered with T2 are the same as those met when incorporating ITI.

In summary, it is very easy to incorporate algorithms with TELA in what concerns static input and measurements that are in advance available in a learning algorithm. What requires more thought, is the measurement of characteristics not supported by the new algorithm. That tends to require programming an interpreter for the output classifier.

Extensive experiments

So far, three major test series constitute the system's main validation test: Those run for this work and reported in the following chapter, those run by Rousu [1996] for his thesis, and those by Elomaa and Rousu [1996b]. Two first test series were run partly simultaneously. Thus, they constitute a test for the multi-user facilities of the system.

TELA has turned out to be just the right tool for executing all these experiments. (Of course that was to be expected, since we had a similar test setup in mind when designing TELA.) Nonetheless, there were some minor complications encountered during the experiments. At first it proved a little hard to adopt to the very strict correctable computing methodology of SEED [Holsti 1989] that now also appears as a part of TELA through its current interface. For instance, if data was saved into a file using TESLA command write and, subsequently, that command was retracted in an attempt to reuse the existing code, then the file written previously as the result of this command would be deleted and a new one would replace it. This is so different from the usual functioning of computer programs that, at first, it will undoubtedly surprise any unfamiliar user. Once we got used to the correctable computing paradigm, things started to work with ease.

There are some things in TELA that could and probably should be changed. For example, more versatile file name extension facility in TESLA would make the system appear much more user friendly, and the often-appearing divide-run-test command sequences in TESLA specifications (see Table 4.1) could easily be abbreviated. Furthermore, a more extensive help facility is required.

Chapter 5

Empirical Evaluation and Validation

This chapter reports on a series of comparative experiments on learning algorithms executed under TELA. The experiments were carried out in order to empirically validate the *Rank* algorithm elaborated in Chapter 3. Furthermore, these experiments act as yet another test case for the correct functioning of TELA. First we outline the general guidelines for the tests: What kind of measurements are taken, and in which domains? What are particular tests designed to reveal? Then we describe briefly the domains, where the comparative measurements are taken. After listing and commenting on the empirical measurements in Section 5.2, the final two sections summarize, analyze, and discuss the results obtained in these common benchmark data sets.

5.1 Experiment setting

In this section we introduce the setting for the empirical experiments. First we sketch a general outline for the tests that are taken. Then we introduce briefly the realizations of the algorithms that are included in the comparison, and describe the most important aspects of the test domains.

5.1.1 Experiment outline

Experimental study is an important and often used method of evaluating learning systems. There is a collection of principles and standard experiment types that have come to be generally accepted goals, means, and methods of evaluating learning systems [Kibler & Langley 1988, Michie *et al*. 1994]. Our experiments conform to these guidelines as far and as completely as the limitations imposed by the wider purpose of this study allow.

There are three different sets of tests reported in the following section. The first one tries to put *Rank* into perspective with respect to other inductive learning programs according to the single most important property of a learning program, the classification accuracy on unseen instances. For this end we run *Rank* on data taken from StatLog project [Michie *et al.* 1994]—a recent comprehensive comparison on inductive learning methods—and evaluate *Rank*'s performance with respect to the reported results. This set of tests consists of three independent experiments.

Once a general impression of *Rank*'s utility is obtained, we want to get a more detailed understanding of how *Rank* manages in comparison with the more established, but similar learning methods. Also, we do not want to observe the algorithms' behavior along one dimension only, but intend to record other characteristic measurements, such as hypothesis size and training time, in addition to prediction accuracy. Furthermore, a good account of the effects of noise on learning is still missing after the first set of tests.

The second set of tests consists of further experiments with uncorrupted data. This time the main emphasis is on measures other than prediction accuracy. Another objective of these experiments is to discover the shape of *Rank*'s learning curve and compare it with the curves of other learning methods. This set of tests consists of five separate experiments.

The last set of tests examines *Rank* and four other programs in the presence of random noise. We try to find out how different noise types affect learning of a hypothesis. The types are attribute and classification noise, and a combination of the two. A "noise curve," recording the degradation pace of an algorithm's prediction accuracy as the noise rate gradually increases, is also registered for the test programs.

Let us already at this point lay down the general guidelines for the experiments and explain how the quality measurements are taken.

Two generally accepted test schemes—both implemented in TESLA—are the basis of these experiments: In cross-validation [Breiman *et al.* 1984, Michie *et al.* 1994] the data is randomly partitioned into a user-specified number of mutually exclusive subsets; one subset is retained for testing and the others are used in training; this procedure is repeated successively for each subset. Thus one is able to utilize the whole available sample as unseen instances in testing as well as in training. The measured values are averages over all subexperiments. The second test scheme partitions the data only into two subsets containing user-specified random portions of the data; one of the subsets is used in training and the other—consisting of instances unseen to the learning algorithm—in testing. We choose the test strategy of each experiment on the basis of literature; i.e., we try to follow the same strategy that was used in earlier studies in order to obtain comparable results. For example, the StatLog experiments were meticulously recorded and the same procedures can be repeated.

Corruption of domains is accomplished using TESLA command randomize, which implements the following strategy [Elomaa & Rousu 1996a]. An attribute is corrupted to noise level n percent by replacing, with probability $n/100$, its original value with a new value drawn randomly from the attribute's range (including the value to be replaced) in all the instances. The probability distribution is uniform: All values in an attribute's range have an equal chance to be selected.

The reported results are always averages over ten repetitions with the same experiment setting. In most cases there is a random element either in the learning program or in the example manipulation. Rerandomization is applied in every repetition. For example, if a random sample is required, then a new sample is drawn randomly at each repetition. The randomization naturally causes variation in the results. Ten repetitions is maybe not enough to guarantee totally reliable results, but it suffices to show the correct level and direction of the results. Moreover, we compensate the low number of repetitions by executing a large number of independent experiments.

Rank only manipulates categorized attribute ranges; it can not handle numerical value ranges. At first this might appear to be a major shortcoming, but that, however, is not the case. Feature manipulation prior to induction is a standard technique that often has to be utilized in practice (see e.g., [Langley 1996]). In these experiments we choose the categorization of numerical attribute value ranges using a separate feature processor before we submit the data to *Rank*. In order to ensure unbiased comparisons with C4.5 we use C4.5 itself as the attribute-categorizing preprocessor. In practice we first run C4.5 on uncategorized data and let it induce some quantization for the attribute value ranges. Then, by using TESLA command map we redefine the data for *Rank* and run it with quantized data.

There are four quality measures monitored in the tests. The first one is the *prediction accuracy* of the classifier produced by a learning program, or more loosely, the prediction accuracy of a program. This refers to the proportion of those instances that are correctly classified by the hypothesis and were not included in the training set. The *time* taken by a program to construct a classifier is the second measure. It does not include the time spent doing basic data set manipulation. Construction times are measured by TELA and expressed in seconds. The third measure is the *rank* of a decision tree as defined in Section 3.1. The *size* of a classifier is the fourth and final measure that is monitored. There are two cases: The size of a decision tree is the total number of (internal and external) nodes in it; the size of a rule set is the sum of the total number of rules in it and the total number of conjuncts in them. Over the years there has been some debate about the correct way to measure these complexities. We have found the above described measures to be at least informative, if not directly comparable.

The results that are reported are always the best ones obtained except for StatLog experiments, where input parameter values for the algorithms were fixed. That is, we vary each algorithm's input parameter values registering only results from the best run. The measure that determines the best result is the prediction accuracy. I.e., the results reported are always results from the run on which the highest classification accuracy was obtained. This causes some additional difficulties in executing the experiments and in interpreting the results, but it is the only fair way to carry out the comparison.

5.1.2 The algorithm implementations

In most of the tests reported below we compare five different induction algorithms; NewID [Boswell 1990b, Tsatsarakis & Sleeman 1993], CN2 [Clark & Niblett 1989, Clark & Boswell 1991], C4.5 [Quinlan 1993], ITI [Utgoff 1994, 1995], and *Rank*. Under the common name *Rank* we actually have three different algorithms: The basic *Findmin*, *Rank*, and its incremental variant *IFM*. Results for these three will be recorded separately. The standard reference of this comparison is a classification heuristic that always predicts the most frequent class among the training examples. We refer to it as the *default* (*class*) method. Clearly, in no situation should the inductive programs decline to do worse than the default class heuristic, if we want to claim that something is gained by using these methods.

NewID is a slight modification of the basic TDIDT method of ID3. The main difference is that no pruning is done while growing the hypothesis tree. Post-pruning of the tree is available upon the user's wish. The split attribute selection method of NewID is not the entropy-based information gain of ID3, but a new method that is based on the Laplacian error estimate [Niblett & Bratko 1986].

We use implementation version 4.1 of the CN2 algorithm [Boswell 1990a], which is a substantially improved variant of the original CN2 rule induction method [Clark & Niblett 1989]. The main difference to the original algorithm is, again, using the Laplacian error estimate instead of entropy-based heuristics. The modifications are explained in detail by Clark and Boswell [1991].

C4.5 is a product that has evolved from Quinlan's studies on decision tree learning over several years. The starting point in the development of C4.5 has been the straightforward TDIDT approach of ID3, but it has progressed a long way. The improvements are explained in detail by Quinlan [1993]. Note that we only use the decision tree producing variant of the algorithm, the possibility to convert trees into rule sets [Quinlan 1987c] is not utilized.

ITI is a modern incremental decision tree learner whose foundation is in the ID5 algorithm [Utgoff 1989], which, in a way, was the incremental equivalent of ID3. Similarly, ITI is intended to be the incremental equivalent of C4.5: It handles continuous attributes more or less the same way as C4.5 does, whereas ID5 is unable to manage them at all. We have incorporated ITI into these experiments mainly to act as a counterpart of *IFM*.

The *Findmin* algorithm in these experiments is the one described in Section 3.2, i.e., the basic *Findmin* extended to handle multivalued variables. (In fact, the extension to handle variables of varying arity from Section 3.5 has also been included.)

The implementation language of *Findmin*, *Rank*, and *IFM* is Ada. The rest of the algorithms have been obtained from their developers; they have all been implemented in C. All algorithms and the TELA environment run on a Sun SPARC workstation environment under `UNIX` operating system.

Table 5.1: Main characteristics of the experiment data. Domains are characterized by the number of attributes (not including the class), average number of values per attribute, number of classes, and total number of examples.

DOMAIN	ATTRIBUTES	VALUES	CLASSES	EXAMPLES
CR.AUST	14	6.1	2	690
VEHICLE	18	7.0	4	846
DIABETES	8	12.5	2	768
SHUTTLE	9	2.9	7	58,000
DNA	60	2.0	3	3,186
MPLX6	11	2.0	2	2,048
LED	7	2.0	10	200
CHESS	36	2.0	2	3,196
TUMOR	17	2.2	22	339
SOYBEAN	35	2.9	15	290
MUSHROOM	22	5.6	2	2,065

5.1.3 Experiment domains

The latest broad interest in machine learning has now lasted just under two decades ever since Feigenbaum's notion of the "knowledge acquisition bottleneck" in expert systems [Feigenbaum 1977] and Michalski's famous soybean disease identification results [Mihalski & Chilausky 1980]. During this time a collection of standard reference data for evaluating the behavior of induction algorithms has accumulated. The literature includes several reports on empirical tests on different algorithms within these standard domains. Hence, to achieve comparable results, we also run tests in some of these domains. It is of utmost importance to experiment with many domains, since the algorithms' behavior tends to depend on the domain characteristics. (For a detailed study on the correlation between algorithms and domain characteristics see Michie *et al*. [1994].) We have selected data sets with varying characteristics in order to evaluate the effects caused by these characteristics. The test data in our experiments comes from two sources: The StatLog project [Michie *et al*. 1994] and the University of California at Irvine repository of machine learning databases [Murphy & Aha 1994].

We now describe briefly the test domains of the experiments. Table 5.1 summarizes the main characteristics of the data sets. A comprehensive description of the data can be found in Appendix B.

CR.AUST: This is one of the most commonly used machine learning databases; it involves assessing credit card applications. The data was introduced to machine learning community by Quinlan [1987c].

VEHICLE: An image data set for testing how 3-dimensional objects can be identified from a 2-dimensional image. Here the images are silhouettes of different types of cars. The data set comes from the StatLog project.

DIABETES: This medical data originates from National Institute of Diabetes and Digestive and Kidney Diseases. It records data gathered from 846 patients. The problem is to predict whether a patient would test positive for diabetes (according to WHO criteria). After categorization the 8 attributes have on average 12.5 values in their range.

SHUTTLE: Data set originating from NASA and concerning the position of radiators within the Space Shuttle. Relatively easy problem that also was one of the StatLog domains. Comes partitioned to training (43,500 examples) and test (14,500 instances) sets. We use this data to test how the test programs scale up with respect to the number of examples.

DNA: Also this data comes partitioned into prespecified training (2,000 examples) and test (1,186 instances) sets. Our version, even though it has 60 attributes, is not the original nucleotide representation [Noordewier *et al.* 1991], but a modification of the binarized version from the StatLog project [Michie *et al.* 1994]. We have modified it by excluding 120 binary attributes, leaving only 60 as the basis of classification. This simplification was suggested by the StatLog group in order to make the problem easier.

MPLX6: This is the six-bit multiplexor function with five irrelevant bits. Multiplexor functions are known to be difficult concepts for standard learning programs to master [Quinlan 1988a]. The data consists of 2048 instances.

LED: The LED display digit identification domain presented by Breiman *et al.* [1984]. The data consists of 200 seven-LED display images representing the decimal digits. There is a 10% attribute noise affecting the examples. The noise makes the data appear inconsistent.

CHESS: A data set describing king and rook versus king and pawn (on a7) chess endgame board positions. This data comes originally from Shapiro's [1983] Ph.D. thesis, where the endgame is analyzed with extreme detail. There are over 3000 examples described by 36 attributes.

TUMOR: Database concerning the location of primary cancer tumor gathered at the University Medical Center, Institute of Oncology, Ljubljana, Slovenia. This is one of the medical domains that has repeatedly appeared in machine learning literature (e.g., [Cestnik *et al.* 1987, Clark & Niblett 1989, Michalski *et al.* 1986, Mingers 1989a]). It is known that four internists (non-specialists) determined a correct location of primary tumor in 32% of cases and four oncologists (specialists) in 42%

of test cases in this domain [Cestnik *et al*. 1987, Michalski *et al*. 1986]. There are 22 classes in this domain.

SOYBEAN: This is the famous database of soybean infection diseases of Michalski and Chilausky [1980]. The examples consist of 35 attributes. There were 19 classes, 4 of which are omitted as suggested by Michalski and Chilausky because they have only few instances. Diagnostic decision rules of 96.2% accuracy for the 15 class case have been acquired from approximately 20 hours of discussions with a plant pathologist [Michalski & Chilausky 1980].

MUSHROOM: This database is another famous and widely applied standard test case for induction algorithms. The data describes mushrooms in terms of their physical characteristics, and classifies them as poisonous or edible. The data set was gathered from books by Fisher [1987] for his Ph.D. work. We use only a random excerpt of 2065 instances of this domain in our experiments.

Of the above listed domains MPLX6 is a special one, since it is extensive, including all the possible value assignments for the attributes, whereas the others are empirically gathered or randomly generated data sets that cover only a part of all the possible instances of the domain. MPLX6 is a Boolean function, LED and CHESS follow man-made exact rules, while the others are governed by the laws of nature.

5.2 Empirical results

This section reports and discusses the empirical results obtained in the experiments outlined in the previous section.

5.2.1 Three StatLog problems

Michie *et al*. [1994] report the prediction accuracies of several learning programs on many different learning problems. We have tabulated part of their results into Table 5.2. For detailed information on the learning programs, the experiments, and the results we refer the reader to the book of Michie *et al*. [1994].

We have results on the StatLog domains for a good collection of contemporary inductive learning methods readily at hand. Thus, we can obtain an overview of *Rank*'s utility by performing the same experiments with it. Learning algorithms that are used in the subsequent experiments are marked with boldface font in Table 5.2. In addition to *Rank* variants, we have added a row for ITI into the table. The programs were originally sorted into relative order in each experiment; the last row records these placings. For the new algorithms would-be placings are given. In the StatLog project the data sets were categorized into groups according to domain types. We have included one data

Table 5.2: Prediction accuracies of twenty-eight learning programs on three of StatLog project's experiments. Column TIME sums the training times over these three experiments and column PLACINGS gives the relative accuracy order of the algorithms in these domains. The test methods are 10-, 9-, and 12-fold cross validation, respectively. The algorithm types are: Stat = statistical method, DT = decision tree learner, R = rule set induction, and NN = neural method.

METHOD	TYPE	CR.AUST	VEHICLE	DIABETES	TIME	PLACINGS
Discrim	Stat	85.9	78.4	77.5	75.5	3, 6, 3
Quadisc	Stat	79.3	85.0	73.8	305.8	21, 1, 11
Logdisc	Stat	85.9	80.8	77.7	809.7	3, 4, 1
SMART	Stat	84.2	78.3	76.8	83,234.2	13, 7, 4
ALLOC80	Stat	79.9	82.7	69.9	2,281.0	19, 3, 21
k-NN	Stat	81.9	72.5	67.6	167.8	15, 11, 22
CASTLE	Stat	85.2	49.5	74.2	95.2	8, 22, 10
CART	DT	85.5	76.5	74.5	122.4	6, 8, 9
IndCART	DT	84.8	70.2	72.9	363.1	10, 16, 14
NewID	**DT**	**81.9**	**70.2**	**71.1**	**42.8**	**15, 16, 19**
AC	DT	81.9	70.4	72.4	7,912.0	15, 15, 18
Baytree	DT	82.9	72.9	72.9	44.7	14, 10, 14
NaiveBay	DT	84.9	44.2	73.8	34.1	9, 23, 11
CN2	**R**	**79.6**	**68.6**	**71.1**	**180.4**	**20, 19, 19**
C4.5	**DT**	**84.5**	**73.4**	**73.0**	**191.5**	**12, 9, 13**
ITrule	R	86.3	67.6	75.5	1,190.1	2, 20, 6
Cal5	DT	86.9	72.1	75.0	284.0	1, 12, 8
Kohonen	NN	–	66.0	72.7	7,928.4	–, 21, 17
DIPOL92	NN	85.9	84.9	77.6	242.0	3, 2, 2
Backprop	NN	84.6	79.3	75.2	22,952.0	11, 5, 7
RBF	NN	85.5	69.3	75.7	1,752.9	6, 18, 5
LVQ	NN	80.3	71.3	72.8	629.4	18, 14, 16
Cascade	NN	–	72.0	–	289.0	–, 13, –
Default	–	**66.0**	**25.0**	**65.0**	–	**22, 24, 23**
ITI	**DT**	**82.0**	**74.0**	**68.3**	**4,295.2**	**15, 9, 22**
Findmin	**DT**	**72.2**	**58.7**	**69.0**	**66.4**	**22, 22, 22**
Rank	**DT**	**85.4**	**70.0**	**75.6**	**69.5**	**8, 18, 6**
IFM	**DT**	**84.1**	**65.3**	**71.3**	**79.0**	**14, 22, 19**

set from each (prediction task) group into this experiment. The test scheme in all these experiments is cross-validation, but each data set has its own folding factor: CR.AUST is tested using 10-fold cross-validation, VEHICLE using 9-fold, and DIABETES using 12-fold. Let us examine these problems in detail one at a time and then present the overall summary.

Cr.Aust domain does not discriminate clearly between different types of prediction methods: Among the best methods there are symbolic and subsymbolic learners as well as statistical discriminators. Furthermore, most methods achieve prediction accuracy that is within seven percentage points. In other words, all methods do more or less equally well on this domain.

None of our test programs belongs to the very peek of best performers in this domain. Indeed, *Rank* turns out to be the best predictor among them. It achieves average accuracy 85.4% using 10-fold cross-validation. That is only 1.5 points less than the best method has recorded in the StatLog experiments and 0.9 points more than C4.5 has scored. *Rank* would have been the eighth best predictor in this task; i.e., it would have belonged to the top third among these algorithms. *IFM* scores slightly less, but is still within the same region. It would have been the number 14 predictor. The basic *Findmin*, which by definition overfits, does substantially worse than its heuristic variants and all other learners. Still it achieves clearly better performance than the default class heuristic does.

Vehicle domain elicits more dispersion into the prediction accuracies of the algorithms. This time statistical and neural methods do on average clearly better than symbolic learners do. One might guess that it is the increased number of classes that causes the confusion; we examine the impact of increased number of classes in detail subsequently. There are a couple of programs that fail even worse than *Findmin* in this domain. *Rank* is slightly better than CN2, about equal with NewID, and somewhat less accurate than C4.5 in this task. It would have placed only as the eighteenth best algorithm. Moreover, *Rank* is now full 15 points behind the best predictor, while the difference was only 1.5 points in the previous experiment. *IFM* falls even further back, but remains clearly more accurate than the basic *Findmin*. ITI now climbs to the top position among our test programs, passing even its original inspiration C4.5.

Diabetes domain, again, does not bring out great variation in prediction accuracies. *Findmin* is almost competitive with the heuristic methods in this task; its accuracy, even though last but one, is only 8.7 percentage units less than that of the best method. Respectively, its heuristic variants—*Rank* and *IFM*—are unable to increase their accuracy much from the base case of *Findmin*. *Rank*, nevertheless, again scores better than C4.5, and is among the best algorithms in this test, placing as the sixth best predictor. *IFM* again does somewhat worse, but remains competitive, surpassing, for instance, NewID and CN2 algorithms.

Overall one cannot draw unambiguous conclusions on what type of predictor should be used in these domains. If we cumulate the prediction accuracies obtained in these three experiments, we see that the best predictor is a neural method, DIPOL92 (248.4%), followed closely by a statistical discriminator, Logdisc (244.4%), and that the best symbolic method, CART (236.5%), is not far behind. *Rank* falls on average 5.8 points behind in prediction accuracy from the best method and most of the difference is due to one domain (VEHICLE). When contrasted with C4.5, *Rank* turns out to be equal on average. If the algorithms were placed in order according to the cumulated accuracies, our test programs would obtain the following placings out of 26: *Rank*—9, C4.5—10, ITI—17, NewID—19, *IFM*—21, CN2—22, *Findmin*—25, and Default—26. This would suggest that our test programs are divided into two groups, *Rank* and C4.5 being clearly more accurate than the rest of the programs. In general, the division is not so clear-cut, as demonstrated by the subsequent experiments.

Column TIME in Table 5.2 shows the cumulated training times as reported by Michie *et al.* [1994]. The training times of *Rank* variants and ITI have been transformed into the same scale according to the times measured for NewID in our own tests. In general, symbolic learners are expedient, the time consumption of statistical methods varies, and neural techniques are slow to train. *Findmin* and its variants belong to the very fastest learners while ITI together with AC turn out to be the slowest symbolic learners in these experiments. The latter two are special algorithms: ITI processes examples one at a time and AC uses look-ahead [Shepherd 1983]. The variants of *Rank* also use look-ahead and *IFM* even processes the examples one at a time. Nonetheless, they still are significantly more expedient on these domains than ITI and AC.

These StatLog problems give us a spectrum of reference for ordering learning programs along one dimension. At the top of the spectrum there are the heavy neural network methods and statistical discriminators with somewhat better performance on average than symbolic learning methods—decision tree and rule learners—have. *Rank* is one of the best members of the latter category. Important differences along other dimensions, such as intelligibility of the resulting classifier, go unnoticed here. These tests hint that increase in the number of classes—the main distinction in the domain characteristics of VEHICLE when opposed to CR.AUST and DIABETES—would be a strongly dispersive factor. Whether that really is the case will be examined subsequently in detail.

Having now positioned the variants of *Rank* into their place in this spectrum, we turn to examine their behavior in more detail. The above comparisons should have made it clear that already at its present composition *Rank* is able to compete with the élite of learning programs in many respects. We restrict ourselves to comparisons of *Rank* and a couple of its equals in the remaining experiments. The exact placement of the following results into the wide spectrum of learning programs is not known, but the above results give us a pretty good idea.

5.2.2 Experiments in noise-free domains

We carry out five more experiments in domains that have no (explicit) noise present. All domains are, though, not error-free; the real-world domains have more or less attribute value recordings missing. This time only seven inductive methods are contrasted with each other. Our first experiment repeats the StatLog problems. We focus on the other quality measurements than prediction accuracy. In the second experiment we test the seven programs on the MPLX6 domain. Quinlan [1988a] has shown that the basic TDIDT approach is inherently unable to come up with the optimal decision tree in this domain. We briefly review how *Rank* and other methods do. The third experiment evaluates the effect of the number of classes, attributes, and attribute values on the different methods. The fourth experiment registers the learning curves of the programs on the VEHICLE domain. We record the prediction accuracies of the programs when the hypothesis is induced from training sets of varying size. Finally we test the algorithms capability to scale up by executing them on the substantially larger SHUTTLE domain.

Subsequently we follow the convention of marking by boldface font, in each separate run, the measurement, which is considered the best among those obtained by the heuristic algorithms (shortest time, lowest rank, smallest hypothesis, and highest accuracy). Emphasizing these values is not intended to disclose anything about the overall performance of the algorithms, but just indicate the best recording.

StatLog domains revisited

Before going into results in other domains, we inspect the values of quality measures other than prediction accuracy recorded by our test programs in the three StatLog domains. Table 5.3 presents the measured values.

From Table 5.3 we can see that C4.5 always prunes the heaviest and, hence, produces the smallest decision trees. Still, *Rank*'s trees are slightly more accurate than those produced by C4.5 on two experiments out of three. Even NewID's trees are smaller than those of *Rank*, in spite that they have higher rank on average. *Findmin* does not prune at all and, therefore, produces trees of excessive complexity. Moreover, its hypotheses have the correct rank, whereas its heuristic variants relax the criterion and always manage to find trees of lower rank. *IFM* comes up with trees that are only slightly more complex than those produced by *Rank*. A further observation is that in the VEHICLE domain the relaxation of *Rank* and *IFM*, even though it reduces the size of their hypotheses to a fourth of that of *Findmin*'s trees, is not enough: They still come up with trees that are much larger than those produced by other algorithms; in tandem, they turn out poor predictors.

Table 5.3: Values of the quality measurements other than accuracy for the seven test programs in the three StatLog domains. Column RK records the average rank of the decision trees produced.

	CR.AUST			VEHICLE			DIABETES		
	TIME	RK	SIZE	TIME	RK	SIZE	TIME	RK	SIZE
Rank	**3.6**	2.0	128.8	20.9	3.0	463.7	**1.8**	**2.0**	188.8
C4.5	4.6	**1.8**	**32.6**	18.2	3.1	**144.8**	5.9	2.4	72.2
NewID	4.0	2.9	118.7	**10.6**	4.0	190.8	3.7	3.9	142.0
CN2	15.1	–	52.6	42.9	–	219.4	6.8	–	**39.8**
IFM	4.8	2.0	173.5	25.5	**2.5**	390.8	3.3	**2.0**	241.0
ITI	236.2	3.3	176.0	1,469.7	3.8	309.5	1,118.1	3.2	311.9
Findmin	10.1	3.0	937.5	15.4	3.0	1737.4	2.9	3.0	1,673.4

The learning times show some variation. When *Rank* succeeds in coming up with a good predictor, it is also the fastest learner among our test programs. However, in the VEHICLE domain in which *Rank* fails, it increases its relative time consumption. As demonstrated analytically, *IFM* has asymptotically the same time requirement as *Rank*; in practice it turns out to be only slightly slower—because of the additional bookkeeping—even though the examples are processed one at a time. A clear contrast to ITI, which uses abundantly more time than its inspiration C4.5. NewID is a bit more expedient than C4.5, and CN2 is relatively slow in all of these experiments. *Findmin*'s time consumption varies, but in general it is surprisingly efficient in comparison with he other methods, considering it uses look-ahead in attribute selection. The real surprise in these domains, however, comes from ITI: It uses orders of magnitude more time than the other methods do. Subsequent experiments demonstrate that this is not always the case: ITI—being an incremental algorithm—is commonly slower than our other test programs, but not that much slower. Let us already at this point offer our view of the reason for ITI's peculiar behavior in this respect. We submit it to the reader that it is continuous attribute value range discretization that hampers ITI's efficiency on these domains. In incremental learning the heavy basic operations (cf. [Fayyad & Irani 1992]) need to be performed over and over again every time a new instance is received. Furthermore, ITI does not use the most modern efficient solutions to the splitting problem [Fulton *et al.* 1995].

Table 5.4: Training times, ranks, and sizes of the hypotheses induced by the seven test programs from three random samples of different size in the six-bit multiplexor function domain. All generated classifiers were 100% accurate. Test strategy is cross-validation.

	2-FOLD			5-FOLD			10-FOLD		
	TIME	RANK	SIZE	TIME	RANK	SIZE	TIME	RANK	SIZE
Rank	14.9	**3.0**	37.0	27.8	**3.0**	36.2	28.4	**3.0**	36.8
C4.5	**1.8**	3.5	49.0	**1.9**	3.2	41.8	**1.7**	3.3	45.6
NewID	3.1	3.5	48.0	3.7	3.6	48.6	3.8	3.3	43.2
CN2	6.2	–	**32.7**	7.7	–	**34.6**	9.5	–	**32.5**
IFM	4.4	**3.0**	40.0	7.4	**3.0**	47.4	8.3	**3.0**	49.6
ITI	20.2	3.5	47.0	45.5	3.4	46.6	52.4	3.3	45.2
Findmin	10.7	3.0	15.0	16.3	3.0	15.0	18.6	3.0	15.0

The multiplexor function

The basic top-down approach of decision tree learning always fails in finding the most concise representation for the multiplexor functions, since the data bits (cf. Appendix B) appear to be more relevant to an instance's classification than the address bits [Quinlan 1988a]. Hence, all pure TDIDT approaches, independent of their split attribute selection heuristics, will produce trees of excessive complexity. Nonetheless, the trees may still be 100% accurate classifiers. Furthermore, the basic TDIDT approach is unable to make any distinction between address bits and irrelevant attributes in the multiplexor function domains [Quinlan 1988a]. This follows from the fact that none of the address bits alone bears any relevance to the classification of an instance. Only combination of address and data bits makes up a feature conveying information about the classification of an instance. In this experiment we examine how our test programs manage the six-bit multiplexor function domain into which we have added five irrelevant attributes.

The experiment consists of three independent runs: The function is induced from three samples of different size. We use 2-, 5-, and 10-fold cross-validation; i.e., the sample sizes are 50%, 80%, and 90% of the total examples of the domain, respectively. Training times, ranks, and sizes of the classifiers thus obtained are listed in Table 5.4. Prediction accuracies are not presented, since all programs, except the default class heuristic, find a 100% accurate classifier at each run. The size of the optimal decision tree for the six-bit multiplexor has 15 nodes. Hence, our first general observation is that *none of the heuristic programs comes very close to the most compact representation; the best results are over twice the size of the optimal decision tree*.

Findmin does not use any heuristic attribute selection and, thus, avoids choosing data bits near to the root of the tree, even though they appear relevant. Therefore, it is always able to find the size-optimal trees. *Rank* and CN2 find the smallest hypotheses of heuristic methods at each run. The former's rank criterion, which is preferred over all other criteria, prohibits it from including any of the irrelevant attributes in the trees produced. The excess with respect to an optimal tree is caused by the fallacious ordering of bits. CN2's success in this respect is due to the better flexibility of rule representation for this task. *IFM* is the only algorithm whose hypothesis size is clearly affected by the increase in sample size. However, considering how it operates, that is to be expected: The additional instances are included into the existing tree, if possible, without increasing its rank, but regardless of the increase in its size. The sample size has no significance to the size of any other algorithm's classifiers.

The time requirement of all test programs, except C4.5, is affected by the increase in sample size. Note that on this domain consisting of only nominal attributes ITI's time consumption reduces to the same order with the other algorithms; though, it remains the slowest learner of them all. An interesting detail is that the incremental approach in *Rank* turns out very profitable in time saving: *IFM* is substantially quicker on this domain than *Rank*. Even *Findmin* can operate faster than *Rank*.

The conclusion of this experiment is that *rank criterion helps Rank to produce more concise decision trees than other algorithms, but even its results are quite far from the optimal result*. However, since *Rank* incorporates the basic *Findmin* as its core, we could obtain the optimal tree by turning all heuristics off. Finally we note that even though the heuristic algorithms *per se* are unable to come up with the most compact representation, quite simple restructuring techniques, like those introduced in IDL by Van de Velde [1990] and applied by Elomaa and Kivinen [1990], can be used to post-process decision trees to reduce their size considerably.

Impact of the number of attribute values

As *Rank* has been developed based on a learning algorithm that can handle only binary classification tasks, it seems quite natural to test how multicategory classifications are managed. The impact of two vs. several classes on an algorithm's success has been a topic of concern with other methods too [Quinlan 1988b]. Also, the number of attributes and the number of values per attribute may have an impact on the return. Absolute figures measuring these effects are hard to come to. Therefore, we explore the relative differences of the learning programs.

Number of classes The first experiment intends to shed some light on the impact of increased number of classes. We execute experiments in the LED, SOYBEAN, and TUMOR domains, which have as many as 10, 15, and 22 classes, respectively. Table 5.5 tabulates the results for these experiments. Our first observation is that *Findmin* cannot tolerate inconsistencies appearing in the data sets LED and TUMOR. The former domain

Table 5.5: Values of the quality measures on the LED, SOYBEAN, and TUMOR domains, which have 10, 15, and 22 classes, respectively. Columns TM and RK contain the average training times and ranks of produced classifiers, respectively. The test strategy in these experiments is 10-fold cross-validation.

	LED, 10 classes				SOYBEAN, 15 classes				TUMOR, 22 classes			
	TM	RK	SIZE	ACC.	TM	RK	SIZE	ACC.	TM	RK	SIZE	ACC.
Rank	**1.1**	**2.0**	**18.4**	63.0	9.4	**2.0**	150.6	85.4	3.5	**2.0**	**42.9**	**43.5**
C4.5	**1.1**	3.0	30.6	69.0	**1.7**	2.9	59.2	83.8	**2.4**	2.9	59.3	41.3
NewID	1.4	3.1	43.6	**73.1**	2.2	3.0	130.8	77.8	3.2	4.0	163.4	37.7
CN2	1.9	–	57.6	66.2	12.2	–	**54.5**	83.7	9.2	–	90.2	42.0
IFM	1.3	**2.0**	21.0	61.3	17.9	**2.0**	175.0	83.6	4.7	**2.0**	46.9	39.9
ITI	3.7	4.0	92.0	68.2	43.9	3.0	79.8	**89.2**	–	–	–	–
Findmin	–	–	–	–	10.3	2.0	208.6	81.4	–	–	–	–
Default	–	0.0	1.0	12.3	–	0.0	1.0	10.6	–	0.0	1.0	26.0

has noise added into it [Breiman *et al.* 1984] and the latter is known to be an inconsistent one because of insufficient attributes. Incidentally, note that the default class heuristic is not necessarily a very good performance reference in this type of multicategory classification task whenever the instances are divided evenly among the classes (cf. the LED domain).

The results lend themselves to many interpretations: Different algorithm scores the best accuracy on each separate domain. Only NewID, whose relative performance deteriorates as the number of classes increases, suffers in a constant manner from the increased number of classes. Anyhow, it is evident that *neither Rank nor IFM suffer from the increased number of classes* more than other heuristic methods do.

On the LED domain *Rank* and *IFM* are fast and construct clearly the smallest hypotheses. Unfortunately they are not very accurate in class prediction. Best accuracy is obtained by NewID. C4.5 and ITI record approximately equal accuracies. The latter, though, using more time and constructing quite large decision trees.

On the SOYBEAN data set ITI, surprisingly, comes up with distinctly the best classifiers, which are both compact and accurate. Of course, the incremental procedure takes more time than batch procedures do. NewID's performance notably deteriorates from the previous experiment: Its hypotheses are considerably less accurate than those of other algorithms. Rank does well, being the second most accurate program, using relatively much time. *IFM* competes head to head with C4.5 and CN2 in prediction accuracy. Again, the look-ahead in *Rank* and *IFM* makes them consume more time than is needed by C4.5 and NewID.

On the last domain of this experiment, TUMOR—an infamous database, *Rank* is able to construct classifiers that are more accurate and concise than those of other methods. The time consumed is a bit more than it does take C4.5 to find its hypothesis. Also *IFM* comes up with smaller decision trees than other methods do; though, they are not quite as accurate as those produced by *Rank*. Even CN2 surpasses C4.5 in accuracy, being the other algorithm, in addition to *Rank*, that is able to obtain the prediction level of medical specialists [Cestnik *et al*. 1987]. NewID continues its decline: The trees produced are neither accurate nor compact. Also ITI stumbles badly; because of some error(s) in the program it is unable to produce a legal decision tree in this task.

All in all, these experiments give evidence contrary to our initial fear that *Rank*'s performance would suffer from the increased number of classes. Indeed, they give the impression that *Rank* rather improves its relative performance than loses any of its advantage when the number of decision categories goes up. However, the evidence is too limited to justify making any far-reaching conclusions.

Number of attributes Similarly as it is hard to manufacture a problem that would watertightly demonstrate the impact of increased number of classes, it is hard to generate a domain for measuring the effects of the number of attributes. Note that we do not mean a case where irrelevant attributes are added to the domain. Such a case produces results that are quite predictable and will hence be disregarded here. We have already run experiments in domains with up to 35 attributes and largely varying average number of values per attribute. We run another experiment to evaluate the effect of the number of attributes, but analyze the meaning of the average number of values on the basis of results that have already been gathered.

Table 5.6 records results from our experiment observing the effect of the number of attributes. Whenever variation is absent from the recorded values, decimals have been omitted from the figures in the table. *Findmin*'s strict matching does not work in the DNA domain, there is some inconsistent element included, which would require relaxed fitting.

In this experiment the domains happened to be relatively easy; in particular, the CHESS and the MUSHROOM domains are mastered almost perfectly by all the programs. On the DNA domain, too, almost equal accuracies are obtained by all the programs. *Findmin*'s and its heuristic variants' time consumption grows considerably when the number of attributes goes up. A result that is consistent with the theory: *Findmin*'s time requirement is polynomially dependent on the number of attributes (Theorem 11). On the other hand, all algorithms increase their training time notably as the number of attributes grows; even those algorithms, which record the least relative increase, double their time consumption when going from domain to another. Classifier sizes tend to grow roughly in the same proportion. The peculiarity of this experiment is that on the DNA domain *IFM* comes up with trees of equal size to those that *Rank* constructs. It does it

Table 5.6: Values of the quality measures on the MUSHROOM, CHESS, and DNA domains, which have 22, 36, and 60 attributes, respectively. Columns TM and R contain the average training times and ranks of produced classifiers, respectively. Test strategy in MUSHROOM and CHESS is 10-fold cross-validation and in DNA 10-time repetition with prespecified training and test sets.

	MUSHROOM, 22 attrs				CHESS, 36 attributes				DNA, 60 attributes			
	TM	R	SIZE	ACC.	TM	R	SIZE	ACC.	TM	R	SIZE	ACC.
Rank	4.7	**1**	10.0	99.7	89.8	**2.0**	119.9	98.8	181.5	**2**	255	91.4
C4.5	**2.3**	2	25.2	**100**	**3.9**	2.5	**51.2**	99.2	**7.5**	3	**103**	**93.9**
NewID	6.6		10.0	99.8	7.6	2.8	65.0	98.7	11.2	3	129	93.0
CN2	19.0	–	10.0	**100**	63.6	–	73.5	98.4	91.8	–	258	91.5
IFM	6.2	**1**	10.0	99.7	149.9	**2.0**	187.1	97.9	44.7	**2**	255	91.9
ITI	26.8	**1**	**9.0**	**100**	129.1	3.0	90.6	**99.7**	457.8	4	285	91.9
Findmin	5.1	1	20.0	100	146.2	2.0	327.7	98.7	–	–	–	–
Default	–	0	1.0	66.6	–	0.0	1.0	52.2	–	0	1	50.8

considerably faster and, furthermore, the trees are even slightly more accurate than those of *Rank*.

By these results and those recorded earlier it is obvious that *the increased number of attributes in a domain does not automatically mean complications to the learning programs*.

Average number of values Finally, in light of the experiments carried out above, we can conclude that *the number of values per attribute in a domain, as such, has no significance to our test programs*. Naturally, even the largest average number of values per attribute in our test domains is quite small. In real-world applications there might be data available with tens if not hundreds of possible values per an attribute. It is evident that in such a domain most of our test programs would suffer heavily. However, it is as evident that either such data should be preprocessed to better suit the algorithms or slight modifications of the algorithms should be developed to better manage such situations.

All in all the conclusion of these experiments has to be that *Rank and IFM tolerate large numbers of classes, attributes, and values per attribute at least as well as other empirical learners, in what concerns classification accuracy*. However, their trees tend to be somewhat larger than those of other methods and it usually also takes more time to construct them. The latter observation could be expected because of the backtracking control procedure of the algorithms. Our objective in the development of *Rank* was to bring it closer to empirical methods, without sacrificing its provable properties, rather

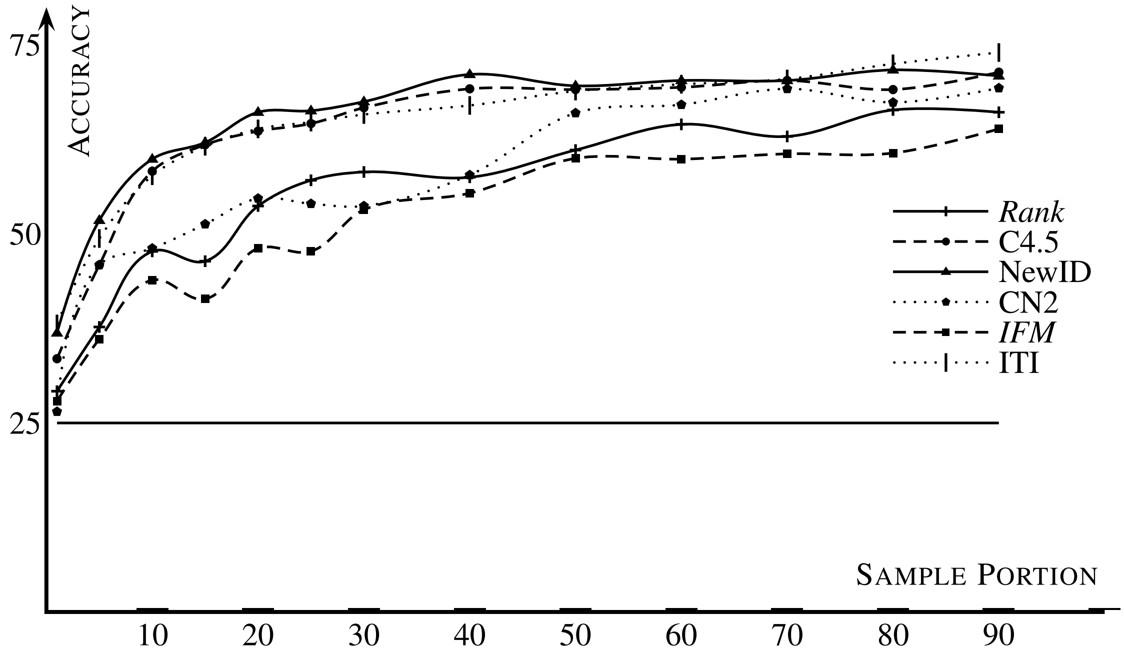


Figure 5.1: Learning curves of the algorithms on a domain concerning prediction of vehicle category.

than to devise yet another purely heuristic algorithm. Therefore, it would be idle speculation to aspire after equivalent results for *Rank* in every respect.

Learning curves

Fig. 5.1 depicts the algorithms' learning curves on the VEHICLE domain. This domain was chosen for this experiment, because it seemed to cause variation to the prediction accuracies; similarly one could expect it to lead to different shaped learning curves. With the clarity of the figure in mind, we have decided to exclude *Findmin* from this experiment. Default class heuristic's accuracy is the horizontal line around 25% accuracy level.

The general appearance of all the learning curves is very similar: Most algorithms reach their final level of accuracy quite rapidly and then settle down. Only small random fluctuation exists in the tail of the curve. Of course, since the programs do not have equal predictive power, the accuracy levels, where the curves settle down, vary. CN2 picks up its final accuracy a little slower than the other programs. Thus its curve is more gently sloping than that of the other algorithms. The curves of *Rank* and *IFM* have more amplified fluctuation in the beginning than those of other algorithms. The reason for their behavior is the rank criterion: The programs resist increasing the rank of their hypotheses and, with small samples, are often able to find acceptable hypothesis trees

Table 5.7: Values of the quality measurements for the seven test programs in the SHUTTLE domain. Test strategy is 10-time repetition with prespecified training and test sets.

	TIME	RANK	SIZE	ACC.
*Rank**	42.3	1	9	98.93
C4.5*	13.8	2	14	99.10
NewID	3,186.3	1	7	99.00
NewID*	41.8	1	9	99.00
CN2	6,670.7	–	39	100
*IFM**	55.7	1	9	99.04
ITI	15,990.7	2	75	99.98
Findmin	–	–	–	–
Default	–	0	1	79.16

with a (too) small rank. Unfortunately, these prove poor predictors. Otherwise none of our test programs exhibits behavior, which differs significantly from that of the others. To our surprise NewID turns out to constantly have the highest prediction accuracy—a phenomenon that has not presented itself in earlier experiments. Though, the curves of C4.5 and ITI are so entangled to that of NewID that one can hardly tell them apart.

As a final note we may draw the reader's attention to the fact that the smallest training set size for which results are recorded, 1% of the domain (8–9 examples), already yields better classifiers than the default class heuristic.

Scaling-up capabilities

The last experiment in this test series examines the algorithms' capability to scale up by running them with a very large database, again, taken from the StatLog project: The SHUTTLE domain contains altogether 58,000 examples. They are divided into a training set of 43,500 examples and a test set of 14,500 examples. The test strategy, of course, is 10-time repetition with these prespecified example sets.

This is a very simple experiment, which only attempts to verify that the test programs can tolerate substantially larger amounts of data than has been used up to this point. The problem itself is pretty easy; almost perfect accuracy for the full data is obtained by a 9-node decision tree [Michie *et al.* 1994, p. 155]. The large domain causes problems to some of our test programs: We were not able to come up with comparable results for all the algorithms. Therefore, we restrain from marking down the best recordings in Table 5.7, which tabulates the measured values obtained in this experiment.

In Table 5.7 the results that have been obtained using categorized data have been marked with an asterisk. For NewID results using categorized and numerical data are given. C4.5 was unable to handle uncategorized numerical data[4]; as usual, the *Rank* variants cannot tolerate numerical attributes. Oddly, CN2 and ITI, for their part, were unable to process the discretized version of the data. Therefore, we cannot present results that are comparable. *Findmin*'s exact fitting, again, fails on this domain, the algorithm is unable to produce a decision tree.

This extreme experiment underlines the importance of expedient handling of numerical attribute value ranges: Time consumptions of the algorithms are on totally different levels depending on whether original or preprocessed data was presented to them; it takes ITI on average almost five hours to come up with a classifier, when C4.5 requires only under 14 seconds on average. For NewID the numerical data requires approximately 80 times the training time of discretized data. On the part of the other quality measurements the results ought to be more comparable. We observe that CN2 is the only algorithm that is able to come up with a perfect classifier. The hypothesis size, however, grows quite large. The other algorithm operating on the numerical data, ITI, also comes up with accurate but large classifier. The rest of the algorithms are within 0.2 percentage points in accuracy. Furthermore, they produce more concise concept representations than CN2 and ITI.

This large domain turns out to be one of the rare cases where *IFM* is able to surpass *Rank* in accuracy. The conclusion of this experiment is ambiguous. We were able to produce a classifier with all our heuristic test programs, but only after modifying the data for some of the algorithms. NewID was the only program capable of handling both versions of the data. However, the size of this domain is already so large, that we find it only reasonable, that the algorithms do not provide for such an amount of examples. The study of *data mining* [Piatetsky-Shapiro 1991] is concerned with discovering general laws from among massive amounts of data.

5.2.3 Experiments in noisy domains

Noise is always a factor in real-world learning situations. We need to conduct further experiments on the impact of unsystematic errors on the algorithms in order to obtain the correct picture and gain proper understanding of *Rank*'s success in this respect. Therefore we carry out this set of tests, which consists of three independent experiments. The first experiment, again, deals with a StatLog problem. This time we artificially corrupt the data of the DIABETES problem to contain 25% random attribute and classification noise. Thus we obtain an initial view of noise's harmfulness to each of the algorithms. The second experiment studies the effect of different types of data corruption on the algorithms. We corrupt the MUSHROOM domain artificially with different types and combinations of noise. The third experiment elicits the algorithms' "noise curves" by

[4] Categorized version of the data has been produced based on the classifier learned by CN2.

Table 5.8: Values of three quality measures on the DIABETES domain with, first, attributes and, then, the class corrupted to noise level 25%. The average ranks of decision trees are not reported, because only minor variations appear in them. Earlier results are repeated for better legibility. Test strategy is 12-fold cross-validation.

	ORIGINAL			ATTRIBUTE NOISE			CLASSIFICATION NOISE		
	TIME	SIZE	ACC.	TIME	SIZE	ACC.	TIME	SIZE	ACC.
Rank	**1.8**	188.8	**75.6**	**1.9**	175.3	70.2	**1.9**	173.5	**66.3**
C4.5	5.9	72.2	73.0	5.5	51.8	**73.1**	10.4	93.3	61.7
NewID	3.7	142.0	71.1	5.1	176.3	68.9	5.7	244.5	61.3
CN2	6.8	**39.8**	71.1	7.2	**31.8**	69.4	6.3	**29.7**	64.8
IFM	3.3	241.0	71.3	2.8	263.7	70.1	3.4	239.4	64.3
ITI	1,118.1	311.9	68.3	1,031.6	303.9	68.0	3,723.5	416.2	57.7
Default	–	1.0	65.0	–	1.0	65.0	–	1.0	63.0

gradually increasing the noise rate of the SOYBEAN domain.

A StatLog problem corrupted

Our first experiment with noise-affected data explores the general impact of noise on the classifiers produced. For this end we re-execute twice the DIABETES problem with the exceptions that this time, first, all attributes and, then, the class labels of instances have been corrupted to the noise level of 25%; i.e., one fourth of values are drawn at random. That should certainly not be a negligible level of noise and should not go unnoticed without having an impact on the measures. This domain's usual test strategy, 12-fold cross-validation, is employed again.

Table 5.8 presents the results obtained in this experiment. *Findmin* is excluded from this and the following experiments, because it is unable to tolerate inconsistencies appearing in the data due to the noise. Therefore it cannot do better than the default class heuristic, into which it ultimately resorts after having tried all rank candidates out.

Table 5.8 shows that attribute noise is handled gracefully by all our test programs: Classifiers remain approximately equal-sized, if not reduce in size, to those learned when no explicit noise was present, training times do not increase significantly, and only reductions of 0–7% in classification accuracy appear. Inconveniently, it is *Rank* that loses the most in accuracy. C4.5 shows excellent capability to tolerate attribute noise: It can even increase its accuracy when noise is introduced into the data.

Further reductions in accuracy come about when attribute noise is changed to classification noise. Moreover, some algorithms increase the size of their classifiers abundantly. C4.5, in particular appears to be sensitive to classification noise: It doubles its training time, increases the hypothesis size, and decreases its accuracy by over 15% from that of the noise-free setting. However, the reason for C4.5 appearing so vulnerable may well be its exceptionally good behavior when dealing with attribute noise on this domain. CN2, on the other hand, accepts classification noise without complications. Also, *Rank* and *IFM* tolerate classification noise well; *Rank* regains its position as the most accurate algorithm among our test programs.

Rank and *IFM* retain their performance throughout the experiment in the sense that they use approximately the same amount of time, the rank of their trees stays intact, and the sizes of these trees are more or less the same independent of whether noise is present or not; only the prediction accuracies vary. This experiment seems to indicate that *Rank*'s and *IFM*'s pruning works better for classification noise than for attribute noise. If this is ascertained in the subsequent experiments, it will not surprise us much, since the algorithms have been designed to take classification noise rather than attribute noise into account. Nevertheless, it would be a minor disappointment, since we have argued that their pruning mechanism ought to be able to take care of other types of noise, too. Let us suspend passing the final judgement until we have the results of the next experiment at hand.

Altogether, this initial experiment would suggest that none of the learning programs is particularly vulnerable neither to attribute nor to classification noise. Because of the straightforward corruption scheme, there are different amounts of randomly drawn values involved in these two experiments. In the former experiment 25% of the values of the eighth attributes of the 768 examples in the test set were randomly drawn. That is, over 1,500 allotted values. In the latter experiment only 25% of the labels of the 768 examples in the test set were drawn at random, i.e., approximately 200 allotted labels. Despite this, consistently with our earlier discussion (Chapter 2), attribute noise is tolerated better than classification noise by all our test programs.

Different types of noise

We have discussed attribute and classification noise and their inseparable combination already on several occasions. Now we test whether our test programs, *Rank* in particular, exhibit different kinds of behavior when facing these noise types. Recall that *Rank* has only been analyzed with classification noise, but is also expected to manage the other noise types. There are two parts in this experiment as well: In the first part the MUSHROOM domain is corrupted with a 20% noise rate using all three noise types. The second part, then, repeats the same experiment with 35% noise rate, this time. Test strategy is 10-fold cross-validation. The bar charts in Fig. 5.2 depict the results of this experiment (for the exact measured figures see Appendix D).

The basis for this experiment was that all the test programs recorded accuracy 100%,

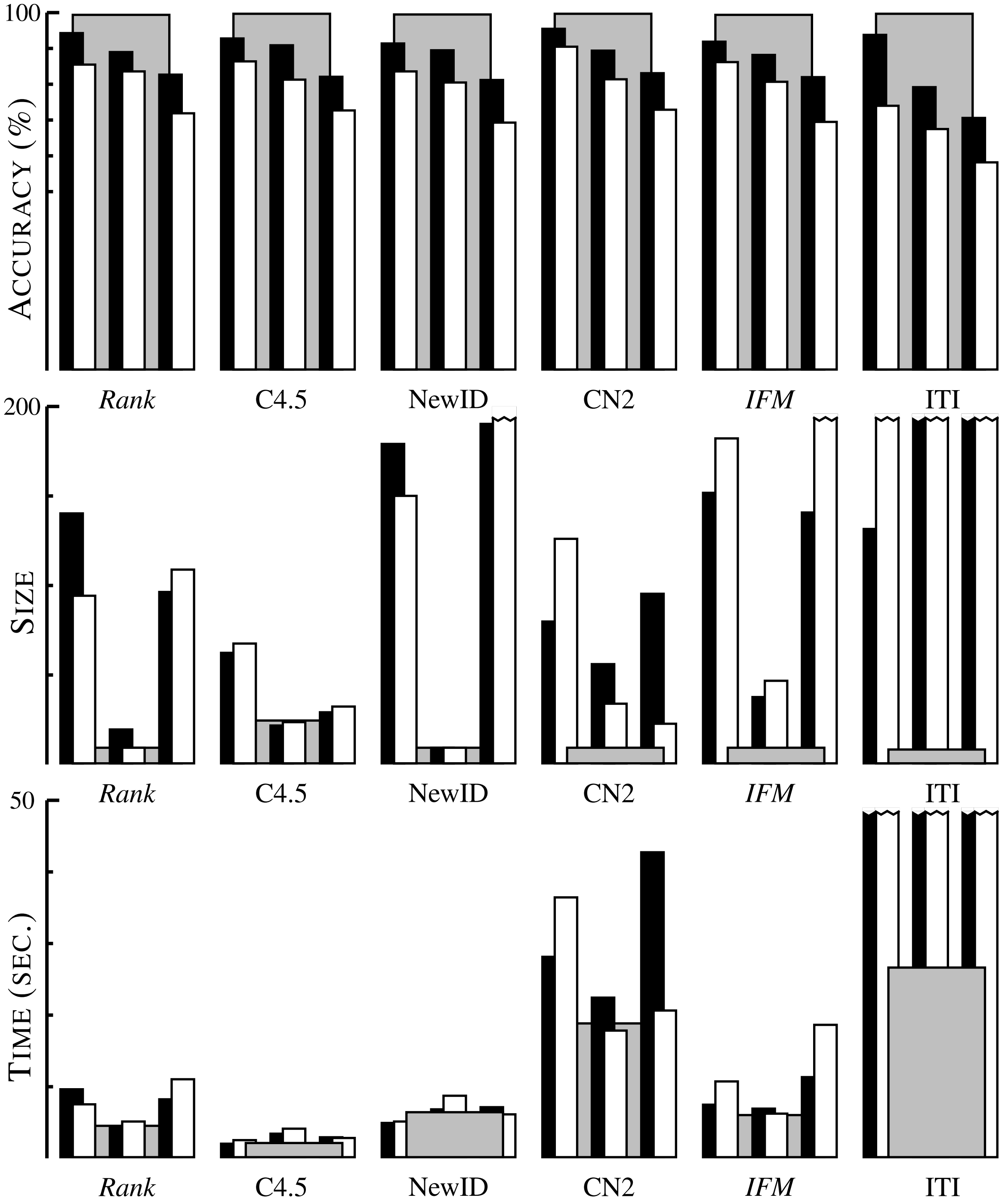


Figure 5.2: The quality measurements on the MUSHROOM domain subjected to noise. For each algorithm three pairs of bars are shown: The first pair depicts measurements under attribute noise, the second one under classification noise, and the third one when both the class label and attribute values are affected by random errors. Black bar presents noise level 20% and the white one noise level 35%. The wide grey bar is the recording obtained for the measurement in question in the noise-free setting.

or slightly less, when no noise was present in the domain (cf. Table 5.6). Furthermore, the small hypotheses were built relatively quickly. The equal original performance of the algorithms lets us observe absolute differences in the algorithms' noise tolerance.

The bars in Fig. 5.2 tell, more or less, the same tale for all of the algorithms: Attribute noise is less damaging for prediction accuracy than classification noise, which in turn proves to be less impairing than the combination of both noise types. Furthermore, the disturbance caused by noise gains power as the noise rate increases; the higher the noise rate, the less accurate the result. The general appearance of the accuracy bars is very similar for all the algorithms. Only ITI is conspicuously more severely hit by noise than other algorithms.

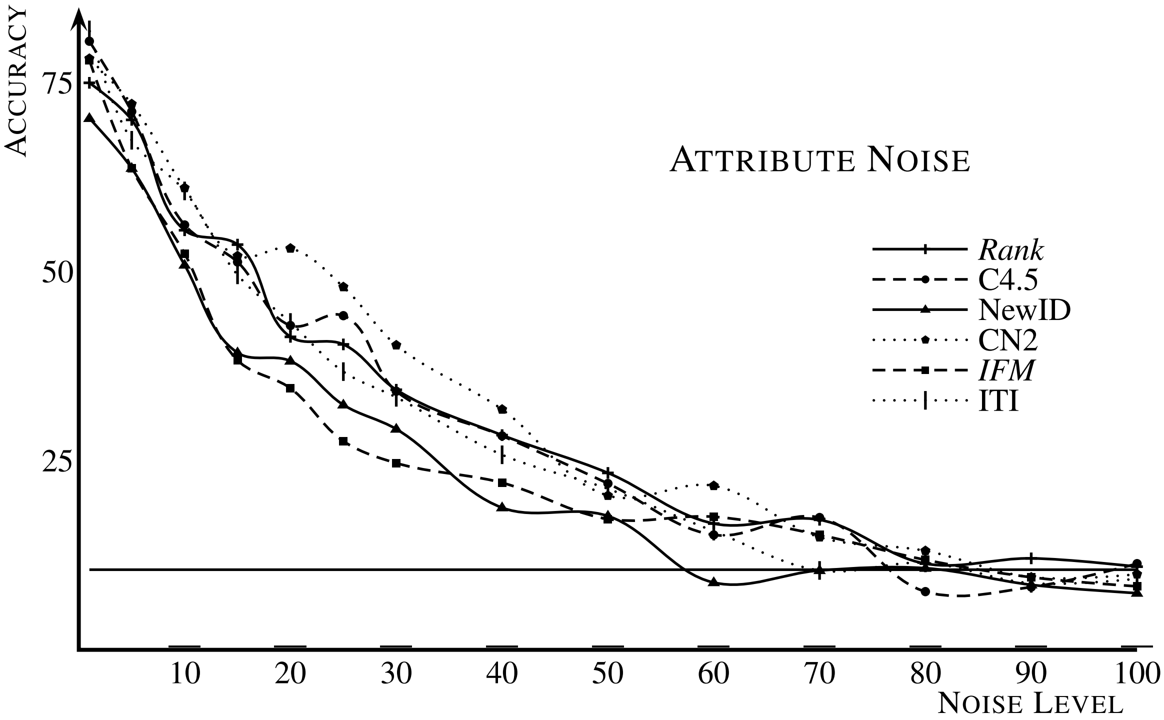
There is more variation present in the bars corresponding to the classifier sizes, but still a common pattern appears: For most algorithms the presence of classification noise alone lets them prune heavier than when attribute noise is prevailing. Otherwise, the algorithms have their individual characteristics as to whether heavier pruning succeeds to keep the resulting classifiers concise as noise rate goes up.

As to the learning times, there seem to be two main lines: In C4.5 and NewID the heavier (post-) pruning, when only classification noise is present, makes them use more time than in the presence of attribute noise. The rest of the algorithms are search intensive (prune on the fly) and require less time when classification noise prevails. In general, there is a natural correspondence between the size of the hypothesis and time spent on constructing it. Finally, ITI again uses untolerable amounts of time and builds trees of excessive size under all types of corruption.

The performance of *Rank* and *IFM* on noisy data accepts the challenge issued by the successful contemporary empirical learners. *Rank*, in particular, is at the same level with the best methods in every respect. *IFM* has been designed to sacrifice compactness on behalf of efficiency. The fact that it also looses somewhat in accuracy is unfortunate, but seems to be inevitable for incremental learners. This experiment does not support the view that *Rank*'s and *IFM*'s pruning would not tolerate other types of noise than classification noise; on the contrary, the two algorithms obtain comparable results with all three corruption schemes. The next experiment should make it clear, whether attribute noise is managed by the *Rank* variants, or not.

Noise curves

Similarly as increasing sample size improves an algorithm's prediction rate steadily, giving thus a "learning curve," increasing the noise rate yields a steadily decreasing prediction accuracy and a "noise curve" for the method can be drawn. This experiment records the noise curves of the test programs in the SOYBEAN domain. Both attribute noise and classification noise values are depicted in Fig. 5.3. Test strategy in this experiment is 10-time repetition: Random portion of 67% of examples is used in training and the remaining 33% is reserved for testing.



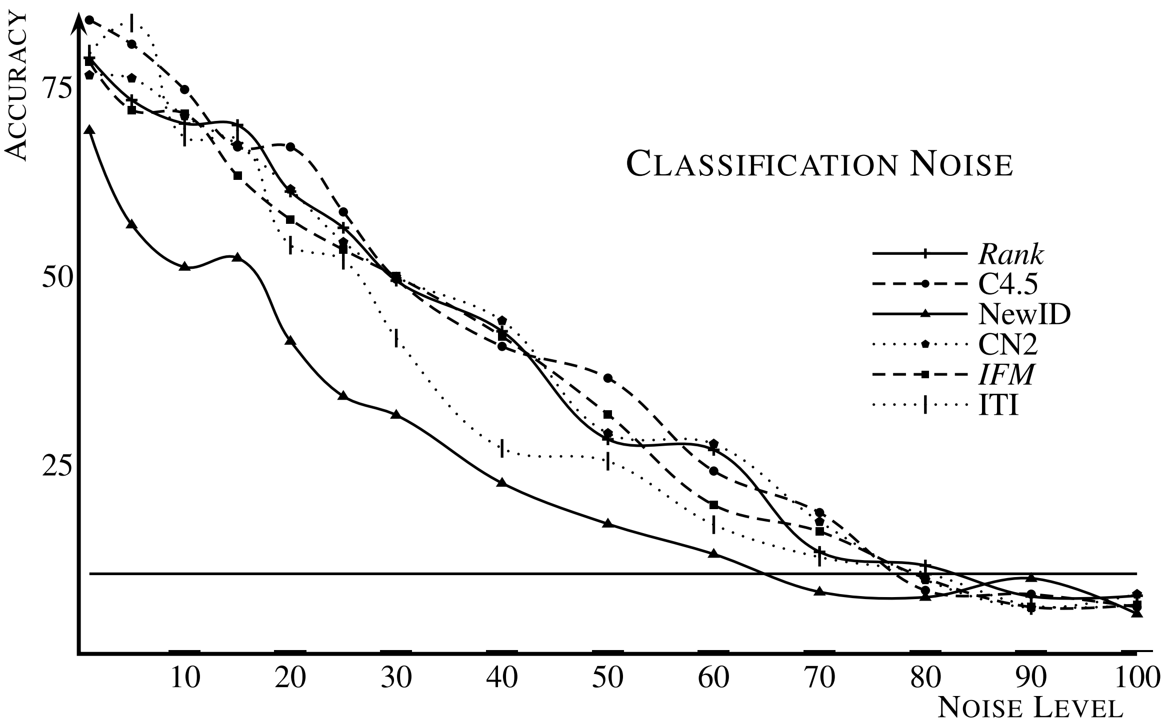


Figure 5.3: The "degradation curves" on a domain concerning the prediction of soybean infections.

Let us analyze these two curves separately. When subjected to attribute noise, the algorithms very rapidly loose their accuracy; already noise level of 1–30%, in general, suffices to halve an algorithm's prediction accuracy on this domain. The decline continues until the accuracy level of the default class is attained. Ultimately all methods resort to predicting the most common class. Hence their accuracy cannot fall significantly below that of the default class heuristic.

The curves of the algorithms get pretty badly tangled together. The only observations that can be safely made is that NewID, as usual, tolerates noise poorly, and that also *IFM*'s prediction accuracy declines faster than that of the other programs; with high noise rates *IFM* catches the other algorithms' degradation pace.

Somewhat surprisingly, when exposed to classification noise, the algorithms seem to do better on this domain. This time most algorithms have a curve that is almost linear; it takes large amounts of noise before the accuracy declines substantially. Again, NewID is discerned as the worst noise handler among our test programs. Also, ITI's poor handling of classification noise, which was already taken notice of in the previous experiment, sets it apart from the other algorithms. The curves of the rest of the algorithms come down head to head.

These curves do not give evidence for the fear of *Rank*'s inferior handling of attribute noise. Rather, these curves demonstrate that *Rank fights both noise types as effectively as other methods do*.

5.3 Summary and analysis of the results

As the first general observation, let us draw the reader's attention to the fact that only ITI ever constructed decision trees with rank beyond value 4 on these experiments concerning several real-world databases. Indeed, the rank of decision trees produced by *Findmin*, *Rank*, and *IFM* never exceeded value 3. We may conclude that typical real-world learning tasks have an accurate decision tree within the smallest rank categories. This ascertains the measurement's utility and supports the idea of learning minimum rank decision trees as a practical approach. Furthermore, it might open up possibilities for developing an even faster and simpler method of learning rank-bounded decision trees for practical purposes.

Incremental methods are, naturally, slower than their off-line equivalents. Processing the examples one at a time, of course, has an overhead. However, as demonstrated by our experiments above, in many real-world domains the difference in practice is quite small. A small increase in running time is gladly accepted in applications demanding on-line learning capability, so long as it stays below times needed by, e.g., the most popular neural methods. Incremental algorithms also tend to produce somewhat larger trees than their batch equivalents. Moreover, their accuracies are on average slightly below those recorded for the trees of batch algorithms. These observations are based on both our pairs of batch and incremental equivalents: C4.5 and ITI as well as *Rank* and *IFM*. Altogether, one has to pay a price for using an incremental algorithm, but since they are special-purpose tools intended to be used only when no alternative exists, that is, in no way, an untolerable tradeoff for the improved applicability.

However, these experiments have clearly shown that ITI does not constitute a competitive incremental alternative to modern inductive learners, mainly because of its slowness, which will invalidate it from many practical tasks. Indeed, *IFM* clearly presents a more serious challenge to batch learners, being only slightly less accurate but efficient in most cases.

There is ample evidence in the above experiments to support the claim that *the overall performance of Rank bears the comparison with its different empirical rivalries*. None of the algorithms, however, is quite modern in the sense that they do not make use of the latest and most advanced general techniques and means developed to aid inductive learning (constructive induction, in particular). All the algorithms are basic experimental methods that could be enhanced with such means, and their performance would improve even substantially.

It is evident that the most important aspect of a learning algorithm is its classification accuracy on unseen instances [Langley 1988, Michie *et al.* 1994]; i.e., its capability to learn the rule governing the classification of instances, if any. However, an important motivation and aim behind the introduction of these tools has been to develop assistants and explanation generators for human experts to use, e.g., in expert system engineering. It is this goal that sets standards to the size and intelligibility of the classifiers produced (in addition to Occam's Razor). If there was not that much difference between some of the algorithms' accuracy, greater gaps in the classifier sizes are detected.

Findmin emphasizes clearly the importance of relaxed fitting in coping with real-world problems. Real-world data is hardly ever perfect. Therefore relaxation in the form of pruning is required even if no significant amounts of noise are present. Of course, this has been taken into account in Valiant's learning framework by allowing ε-error and δ-uncertainty.

The feeling concerning the relative performance of the programs obtained from these experiments is that C4.5, *Rank*, and CN2 are pretty equal in prediction accuracy. C4.5, however, is on average clearly faster than the two, and constructs more concise hypotheses. NewID is distinctly poorer learner than the three. *IFM* can compete with it in every respect. ITI still has many deficiencies, which make it unable to compete with the other algorithms. Nevertheless, all heuristic algorithms give clear advantage over choosing the simple default class heuristic.

Let us conclude this section with the following piece of information. Summing over all different tabulated results, the average accuracies of the test programs are:

C4.5	81.11%
Rank	80.95%
CN2	80.55%
NewID	79.43%
IFM	79.42%
ITI	79.12%
Default	53.79%

This result does not necessarily give a fair picture of the programs' actual utility, since the test domains are not representative of those encountered in practice. On the contrary, the test domains are often such that they are expected to cause difficulties to our test programs. All in all, even though clear differences in the algorithms' utility can be observed in particular experiments, only fairly small differences exist when taking the average over all test domains (cf. [Elomaa 1994]).

5.4 Discussion

Our empirical tests have succeeded in verifying that *Rank* is a highly competitive decision tree learning technique and that TELA is a valuable tool for experiment design and execution. However, both TELA and *Rank* have many natural development opportunities that would enhance their utility and user-friendliness.

The most obvious deficiency in *Rank* is its time consumption's dependence on the number of variables in the domain, which makes the algorithm unstable in this respect. Even though *Rank* never loses its grip similarly as ITI does, it would be desirable to attain a better efficiency. To relax the dependency of the number of variables would require changing the learning approach profoundly. However, there exists an implementation technique that would enhance *Rank*'s speed substantially: Because of the multirecursive operation of the algorithm, dynamic programming would bring substantial savings into its time consumption. Furthermore, tabulating intermediate results would open up possibilities to similar optimizations that are implemented in *IFM*: Avoiding unnecessary recursive calls to *Find*.

We have evaluated *Findmin*, *Rank*, and *IFM* in separation, even though they are implemented all together. Only seldom were there situations where using *Findmin* or *IFM* would have proved more profitable than using *Rank*, but still a couple of occasions exist. Combined these three variants of *Findmin* would have been an even more successful decision tree learner than *Rank* alone. However, we do not have a way of knowing when to use which variant. Improving the adaptivity of *Rank* remains a topic for future research.

Chapter 6

Conclusion

We conclude this dissertation by, first, summarizing the work that was reported and, then, presenting some remarks about the work: What could be done to continue the work? Which topics were omitted and how important are they? What are the practical and principal implications of this work? How does it relate to current trends in machine learning research?

6.1 Summary

After the introductory chapters, in Chapter 3, the first original contribution of this dissertation was presented. Based on the decision tree learning algorithm *Findmin* of Ehrenfeucht and Haussler [1989] a new empirical decision tree learner, *Rank*, with firm theoretical underpinnings was developed. All results concerning the algorithm's functionality were demonstrated in the framework of multivalued variables and classes, which is the normal state of affairs in the application domains of learning algorithms. (It can be argued that multivalued nominal attributes can be quite easily converted into Boolean ones, but since intelligibility is one of the strongest advantages of decision trees, all such changes are damaging in practice.) In order to cope with dynamic changes in the application domains, results of incremental learning of rank-bounded decision trees were demonstrated. Lastly, the unclinical nature of real world was taken into account by providing for random errors in the classification of training examples. Together these three modifications to the original algorithm entail practicability.

In Chapter 4 the need of an integrated testbed for learning algorithms was demonstrated and the design of such an environment, TELA, was presented. TELA has many advanced features: It incorporates an experiment specification language that forces the user to follow strict rules in experiment design, it has an interactive user interface with incremental execution facility, semiautomatic algorithm inclusion is supported, automatic collection of results is the quintessence of the platform. TELA has been designed so that minimum amount of knowledge of a learning program is required before it can be used with TELA. It also tries to provide as much assistance as possible for the actual inclusion of a learning algorithm. The system relieves its user from many simple but tedious duties that are unavoidable in testing learning algorithms.

A central feature of TELA is the support provided for experiment design and incremental execution of specifications. Being able to design test sequences on a higher level is essential for the reliability of the tests and comparisons. Moreover, experiments designed with a carefully elaborated specification language like TESLA are more likely to turn out to be useful than *ad hoc* experiments. Finally, experimentation is unavoidably iterative by nature: The results determine whether a test sequence has been successful or whether it has to be re-executed. In TELA incremental execution of TESLA specifications is supported by the SEED interface.

TELA is still a prototype system, an initial attempt towards a more ambitious environment. In order to gather as much feedback as possible TELA is distributed freely to all interested parties. Even though TELA has been tested extensively, it is yet to confront the real challenge: Application in a real-world development process. Moreover, numerous extension possibilities exist; only experience gathered from different projects and users will tell what should be the final form of the system.

In Chapter 5 the algorithm *Rank* and the environment TELA were compared together with several contemporary inductive algorithms on a large number of real-world learning domains. Several different aspects of the algorithms and their hypothesis classifiers were examined in these experiments. Initially we provided a framework for the subsequent experiments by running all our test programs in three data sets from the StatLog project [Michie *et al*. 1994], which has related the performance of empirical learning algorithms with respect to statistical discriminators and neural networks. The second group of experiments was run on domains with varying characteristics, but with no noise present. In these experiments we emphasized other quality measurements in addition to the prediction accuracy. We explored the algorithms' capability to tend towards an optimal decision tree; their capability to tolerate increases in the number of classes and attributes, and in the average number of values per attribute. Learning curves were recorded in one domain. Finally, the algorithms' capability to scale up was examined by running them in a substantially larger domain than those that had been used up to that point. The final set of experiments concentrated on the effects of noise. Different combinations of noise and their impact on the classifiers produced was examined in three experiments. All in all, *Rank* is successful in these tests: It attains the overall performance of C4.5, which is considered one of the best decision tree learners of today.

The results of this thesis include several individual technical contributions. Two major constructive contributions come out of this research: The successful decision tree learner *Rank* and the uniquely useful testing environment TELA. Our empirical evaluation constitutes an important addition to the ever-increasing volume of knowledge obtained by empirical experimentation. The main contribution of this thesis, however, lies in the development process of *Rank*: The work comprises a successful and extensive case study of the theoretical design rationale in the development of an empirical learning algorithm.

6.2 Remarks

Now that learning decision trees has proven to be intractable in practice [Hancock *et al*. 1995], there is no other alternative than to tighten the syntactic restrictions posed to the resulting hypothesis tree somehow. Delimiting the rank of a decision tree, naturally, is not the only way, but as demonstrated in our empirical work, it is one that has high relevance in real-world domains.

There are many routes that could be followed to continue this work. It is the easiest task to come up with further theoretical results that could be proved and practical improvements that could be implemented.

On the theoretical side many basic results could be worked on. As an example, many other noise models than that which was used in this work could be considered. We touched briefly the on-line learning model of Littlestone [1988]; the analysis of *IFM* under this model ought to be carried through so that comparison with Simon's [1995] work can be made. Analysis of *Rank* could be carried out in a more realistic learning model than the PAC model. For instance, the *Agnostic* PAC model [Kearns *et al*. 1992] might be a better alternative as advocated by Maass [1994].

The theoretical underpinnings of our work have attracted mild criticism [Auer *et al*. 1995]: It has been claimed that the classification noise model does not conform to those situations that are encountered in the real world, which then would lead to inferior performance in practice. However, the noise handling technique that we end up with is exactly the same that was originally present in the ID3 algorithm [Quinlan 1983]. Furthermore, our empirical tests verify that the pruning incorporated into *Rank* works without complications in all of the real-world domains tried.

The practical improvement possibilities of *Rank* include, among others, the following details. The fact that *Rank* is not able to handle numerical attribute ranges is the factor that most clearly sets it apart from the rest of our test programs. However, the general solution to this problem [Fayyad & Irani 1993, Fulton *et al*. 1995] could easily be incorporated into *Rank*. Furthermore, we are aware of the optimal solution to the problem [Elomaa & Rousu 1996b]; it would enhance the method's efficiency substantially. How to reconcile the method elegantly with the minimum rank criterion, of course, remains a problem. Enhancing *Rank*'s parameter independence is a topic for future research. At its present composition the algorithm is quite sensitive to the selection of input parameter values. Furthermore, we would like it to be automatic that the correct (best for the task at hand) *Findmin* variant is chosen. For TELA several improvement proposals were given already in Chapter 4. Work for implementing some of these ideas has already commenced.

Of course, further tests with both *Rank* and TELA are always welcome. Even though both programs have been tested extensively, new empirical experiments are liable to expose further strengths and weaknesses in *Rank* and TELA. Moreover, TELA still lacks application in a real-world development process, which only will give the ultimate validation to the system's utility.

References

AIELLO, W. & MIHAIL, M. [1991]. "Learning the Fourier spectrum of probabilistic lists and trees." In *Proc. Second Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 291–9). ACM, New York, NY & SIAM, Philadelphia, PA.

AMSTERDAM, J. [1988a]. "Extending the Valiant learning model." In J. Laird (ed.), *Proc. Fifth International Conference on Machine Learning* (pp. 381–94). Morgan Kaufmann, San Mateo, CA.

——— [1988b]. "Some philosophical problems with formal learning theory." In *Proc. Seventh National Conference on Artificial Intelligence* (pp. 580–4). Morgan Kaufmann, San Mateo, CA.

ANGLUIN, D. [1988]. "Queries and concept learning." *Mach. Learn.* **2**, 319–42.

——— [1992]. "Computational learning theory: Survey and selected bibliography." In *Proc. Twenty-Fourth Annual ACM Symposium on Theory of Computing* (pp. 351–69). ACM Press, New York, NY.

ANGLUIN, D. & LAIRD, P. [1988]. "Learning from noisy examples." *Mach. Learn.* **2**, 343–70.

ANTHONY, M. & BIGGS, N. [1992]. *Computational Learning Theory*. Cambridge University Press, Cambridge.

ARBAB, B. & MICHIE, D. [1985]. "Generating rules from examples." In *Proc. Ninth International Joint Conference on Artificial Intelligence* (pp. 631–3). Morgan Kaufmann, Los Altos, CA.

AUER, P., HOLTE, R. & MAASS, W. [1995]. "Theory and application of agnostic PAC-learning with small decision trees." In A. Prieditis & S. Russell (eds.), *Proc. Twelfth International Conference on Machine Learning* (pp. 21–9). Morgan Kaufmann, San Francisco, CA.

BECKS, A., BJRKSTEN, M., HYVRINEN, I., NYMAN, M. & TIMONEN, T. [1992]. TELA. Rep. C-1992-43. Department of Computer Science, University of Helsinki. (In Finnish).

BLUM, A. [1992]. "Rank-r decision trees are a subclass of r-decision lists." *Inf. Process. Lett.* **42**, 183–5.

BLUMER, A., EHRENFEUCHT, A., HAUSSLER, D. & WARMUTH, M. [1987]. "Occam's Razor." *Inf. Process. Lett.* **24**, 377–80.

——— [1989]. "Learnability and the Vapnik-Chervonenkis dimension." *J. ACM* **36**, 929–65.

BOARD, R. & PITT, L. [1992]. "On the necessity of Occam algorithms." *Theor. Comput. Sci.* **100**, 157–84.

BOHANEC, M. & BRATKO, I. [1994]. "Trading accuracy for simplicity in decision trees." *Mach. Learn.* **15**, 223–50.

BOSWELL, R. [1990a]. Manual for CN2 version 4.1. Rep. TI/P2154/RAB/4/1.3. The Turing Institute, Glasgow.

——— [1990b]. Manual for NewID version 4.1. Rep. TI/P2154/RAB/4/2.3. The Turing Institute, Glasgow.

BOUCHERON, S. & SALLANTIN, J. [1988]. "Learnability in the presence of noise." In D. Sleeman (ed.), *Proc. Third European Working Session on Learning* (pp. 25–35). Pitman, London.

BREIMAN, L., FRIEDMAN, J., OLSHEN, R. & STONE, C. [1984]. *Classification and Regression Trees*. Wadsworth, Pacific Grove, CA.

BUNTINE, W. [1989]. "A critique of the Valiant model." In *Proc. Eleventh International Joint Conference on Artificial Intelligence* (pp. 837–42). Morgan Kaufmann, San Mateo, CA.

——— [1990]. "Myths and legends in learning classification rules." In *Proc. Eighth National Conference on Artificial Intelligence* (pp. 736–42). MIT Press, Cambridge, MA.

BUNTINE, W. & CARUANA, R. [1993]. Introduction to IND version 2.1 and recursive partitioning. Rep. FIA-93-03. Artificial Intelligence Research Branch, NASA Ames Research Center.

BUNTINE, W. & NIBLETT, T. [1992]. "A further comparison of splitting rules for decision-tree induction." *Mach. Learn.* **8**, 75–85.

CARBONELL, J., MICHALSKI, R. & MITCHELL, T. [1983]. "An overview of machine learning." In R. Michalski, J. Carbonell & T. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach* (pp. 3–23). Tioga, Palo Alto, CA.

CARTER, C. & CATLETT, J. [1987]. "Assessing credit card applications using machine learning." *IEEE Expert* **2**(3), 71–9.

CATLETT, J. [1991]. "Megainduction: A test flight." In L. Birnbaum & G. Collins (eds.), *Proc. Eighth International Workshop on Machine Learning* (pp. 596–9). Morgan Kaufmann, San Mateo, CA.

CAUSSE, K., SIMS, P., MORIK, K. & ROUVEIROL, C. [1990]. "A comparative study of the representation languages used in the Machine Learning Toolbox." In *Proc. Annual ESPRIT Conference* (pp. 326–41). Kluwer, Norwell, MA.

CESTNIK, B., KONONENKO, I. & BRATKO, I. [1987]. "Assistant 86: A knowledge-elicitation tool for sophisticated users." In I. Bratko & N. Lavrač (eds.), *Progress in Machine Learning, Proc. Second European Working Session on Learning* (pp. 31–45). Sigma Press, Wilmslow, UK.

CHOU, P. [1991]. "Optimal partitioning for classification and regression trees." *IEEE Trans. Pattern Anal. Mach. Intell.* **4**, 340–54.

CLARK, P. & BOSWELL, R. [1991]. "Rule induction with CN2: Some recent improvements." In Y. Kodratoff (ed.), *Proc. Fifth European Working Session on Learning*, Lecture Notes in Computer Science **482** (pp. 151–63). Springer-Verlag, Berlin.

CLARK, P. & NIBLETT, T. [1989]. "The CN2 induction algorithm." *Mach. Learn.* **3**, 261–83.

COHEN, W. [1993]. "Efficient pruning methods for separate-and-conquer rule learning systems." In *Proc. Thirteenth International Joint Conference on Artificial Intelligence* (pp. 988–94). Morgan Kaufmann, San Mateo, CA.

COMER, D. & SETHI, R. [1977]. "The complexity of trie index construction." *J. ACM* **24**, 428–40.

DIETTERICH, T. [1989]. "Limitations on inductive learning." In A. Segre (ed.), *Proc. Sixth International Workshop on Machine Learning* (pp. 124–8). Morgan Kaufmann, San Mateo, CA.

DUDA, R. & HART, P. [1973]. *Pattern Classification and Scene Analysis*. John Wiley & Sons, New York, NY.

EHRENFEUCHT, A. & HAUSSLER, D. [1989]. "Learning decision trees from random examples." *Inf. Comput.* **82**, 231–46.

ELOMAA, T. [1992]. A hybrid approach to decision tree learning. Rep. C-1992-61. Department of Computer Science, University of Helsinki.

——— [1994]. "In defense of C4.5: Notes on learning one-level decision trees." In W. Cohen & H. Hirsh (eds.), *Machine Learning: Proc. Eleventh International Conference* (pp. 62–9). Morgan Kaufmann, San Francisco, CA.

ELOMAA, T., HOLSTI, N. & HYVRINEN, I. [1995]. "TELA: A platform for experimenting with attribute-based learning programs." In A. Aamodt & J. Komorowski (eds.), *Proc. Fifth Scandinavian Conference on Artificial Intelligence* (pp. 391–5). IOS Press, Amsterdam & Ohmsha Ltd., Tokyo.

ELOMAA, T. & KIVINEN, J. [1990]. "On inducing topologically minimal decision trees." In *Proc. Second International IEEE Conference on Tools for Artificial Intelligence* (pp. 746–52). IEEE Computer Society Press, Los Alamitos, CA.

——— [1991]. Learning decision trees from noisy examples. Rep. A-1991-3. Department of Computer Science, University of Helsinki.

ELOMAA, T. & ROUSU, J. [1996a]. Attribute-based induction and TELA. Rep. in preparation. Department of Computer Science, University of Helsinki.

——— [1996b]. Finding optimal multi-splits for numerical attributes in decision tree learning. NeuroCOLT Tech. Rep. NC-TR-96-041. Department of Computer Science, Royal Holloway, University of London.

ELOMAA, T. & UKKONEN, E. [1994]. "A geometric approach to feature selection." In F. Bergadano & L. De Raedt (eds.), *Machine Learning: ECML-94, Proc. Seventh European Conference on Machine Learning*, Lecture Notes in Computer Science **784** (pp. 351–4). Springer-Verlag, Berlin.

FAYYAD, U. & IRANI, K. [1992]. "On the handling of continuous-valued attributes in decision tree generation." *Mach. Learn.* **8**, 87–102.

——— [1993]. "Multi-interval discretization of continuous-valued attributes for classification learning." In *Proc. Thirteenth International Joint Conference on Artificial Intelligence* (pp. 1022–7). Morgan Kaufmann, San Mateo, CA.

FEIGENBAUM, E. [1977]. "The art of artificial intelligence: I. themes and case studies of knowledge engineering." In *Proc. Fifth International Joint Conference on Artificial Intelligence* (pp. 1014–29). MIT, Cambridge, MA.

FENG, C., KING, R., SUTHERLAND, A., MUGGLETON, S. & HENERY, R. [1994]. "Symbolic classifiers: Conditions to have good accuracy performance." In P. Cheeseman & R. Oldford (eds.), *Selecting Models from Data, Artificial Intelligence and Statistics IV*, Lecture Notes in Statistics **89** (pp. 371–80). Springer-Verlag, New York.

FISHER, D. [1987]. Knowledge acquisition via incremental conceptual clustering. Ph. D. Thesis, Rep. 87-22. Department of Information and Computer Science, University of California at Irvine.

FISHER, D. & SCHLIMMER, J. [1988]. "Concept simplification and prediction accuracy." In J. Laird (ed.), *Proc. Fifth International Conference on Machine Learning* (pp. 22–8). Morgan Kaufmann, San Mateo, CA.

FRIEDMAN, J. [1977]. "A recursive partitioning decision rule for nonparametric classification." *IEEE Trans. Comput.* **26**, 404–8.

FULTON, T., KASIF, S. & SALZBERG, S. [1995]. "Efficient algorithms for finding multi-way splits for decision trees." In A. Prieditis & S. Russell (eds.), *Proc. Twelfth International Conference on Machine Learning* (pp. 244–51). Morgan Kaufmann, San Francisco, CA.

GELFAND, S., RAVISHANKAR, C. & DELP, E. [1991]. "An iterative growing and pruning algorithm for classification tree design." *IEEE Trans. Pattern Anal. Mach. Intell.* **13**, 163–74.

HANCOCK, T. [1990]. "Identifying μ-formula decision trees with queries." In M. Fulk & J. Case (eds.), *Proc. Third Annual Workshop on Computational Learning Theory* (pp. 23–37). Morgan Kaufmann, San Mateo, CA.

——— [1991]. "Learning 2μDNF formulas and $k\mu$ decision trees." In L. Valiant & M. Warmuth (eds.), *Proc. Fourth Annual Workshop on Computational Learning Theory* (pp. 199–209). Morgan Kaufmann, San Mateo, CA.

——— [1993]. "Learning $k\mu$ decision trees on the uniform distribution." In L. Pitt (ed.), *Proc. Sixth Annual ACM Conference on Computational Learning Theory* (pp. 352–60). ACM Press, New York, NY.

HANCOCK, T., JIANG, T., LI, M. & TROMP, J. [1995]. "Lower bounds on learning decision lists and trees." In E. Mayr & C. Puech (eds.), *Proc. Twelfth Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science **900** (pp. 527–38). Springer-Verlag, Berlin.

HAUSSLER, D. [1988]. "Quantifying inductive bias: AI learning algorithms and Valiant's learning framework." *Artif. Intell.* **36**, 177–221.

HAUSSLER, D., KEARNS, M., LITTLESTONE, N. & WARMUTH, M. [1991]. "Equivalence of models for polynomial learnability." *Inf. Comput.* **95**, 129–61.

HENRICHON, E. & FU, K.-S. [1969]. "A nonparametric partitioning procedure for pattern classification." *IEEE Trans. Comput.* **18**, 614–24.

HOEFFDING, W. [1963]. "Probability inequalities for sums of bounded random variables." *J. Am. Statistical Assoc.* **58**, 13–30.

HOLSTI, N. [1989]. "A session editor with incremental execution functions." *Softw. Pract. Exper.* **19**, 329–50.

HOLTE, R. [1993]. "Very simple classification rules perform well on most commonly used data sets." *Mach. Learn.* **11**, 63–90.

HYAFIL, L. & RIVEST, R. [1976]. "Constructing optimal binary decision trees is NP-complete." *Inf. Process. Lett.* **5**, 15–7.

IBA, W. & LANGLEY, P. [1992]. "Induction of one-level decision trees." In D. Sleeman & P. Edwards (eds.), *Machine Learning: Proc. Ninth International Workshop* (pp. 233–40). Morgan Kaufmann, San Mateo, CA.

IBA, W., WOGULIS, J. & LANGLEY, P. [1988]. "Trading off simplicity and coverage in incremental concept learning." In J. Laird (ed.), *Proc. Fifth International Conference on Machine Learning* (pp. 73–9). Morgan Kaufmann, San Mateo, CA.

KALKANIS, G. & CONROY, G. [1991]. "Principles of induction and approaches to attribute based induction." *The Knowl. Eng. Rev.* **6**, 307–33.

KEARNS, M. & LI, M. [1988]. "Learning in the presence of malicious errors." In *Proc. Twentieth Annual ACM Symposium on Theory of Computing* (pp. 267–80). ACM Press, New York, NY.

KEARNS, M., SCHAPIRE, R. & SELLIE, L. [1992]. "Towards efficient agnostic learning." In *Proc. Fifth Annual ACM Workshop on Computational Learning Theory* (pp. 341–52). ACM Press, New York, NY.

KEARNS, M. & VAZIRANI, U. [1994]. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA.

KIBLER, D. & LANGLEY, P. [1988]. "Machine learning as an experimental science." In D. Sleeman (ed.), *Proc. Third European Working Session on Learning* (pp. 81–92). Pitman, London.

KNUTH, D. [1969]. *The Art of Computer Programming 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA.

KOHAVI, R. [1994]. "Bottom-up induction of oblivious read-once decision graphs." In F. Bergadano & L. De Raedt (eds.), *Machine Learning: ECML-94, Proc. Seventh European Conference on Machine Learning*, Lecture Notes in Computer Science **784** (pp. 154–69). Springer-Verlag, Berlin.

KONONEKO, I. [1993]. "Successive naive Bayesian classifier." *Informatica* **17**, 167–74.

KONONENKO, I. & BRATKO, I. [1991]. "Information-based evaluation criterion for classifier's performance." *Mach. Learn.* **6**, 67–80.

KUSHILEVITZ, E. & MANSOUR, Y. [1991]. "Learning decision trees using the Fourier spectrum." In *Proc. Twenty-Third Annual ACM Symposium on Theory of Computing* (pp. 455–64). ACM Press, New York, NY.

LAIRD, P. [1988]. *Learning from Good and Bad Data*. Kluwer, Norwell, MA.

LAMMINJOKI, T. [1995]. Personal communication.

LANDEWEERD, G., TIMMERS, T., GELSEMA, E., BINS, M. & HALIE, M. [1983]. "Binary tree versus single level tree classification of white blood cells." *Pattern Recogn.* **16**, 571–7.

LANGLEY, P. [1988]. "Machine learning as an experimental science." *Mach. Learn.* **3**, 5–8.

——— [1996]. *Elements of Machine Learning*. Morgan Kaufmann, San Francisco, CA.

LINIAL, N., MANSOUR, Y. & NISAN, N. [1989]. "Constant depth circuits, Fourier transform, and learnability." In *Proc. Thirtieth Annual IEEE Symposium on Foundations of Computer Science* (pp. 574–9). IEEE Computer Society Press, Los Alamitos, CA.

LITTLESTONE, N. [1988]. "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm." *Mach. Learn.* **2**, 285–318.

MAASS, W. [1994]. "Efficient agnostic PAC-learning with simple hypotheses." In *Proc. Seventh Annual ACM Conference on Computational Learning Theory* (pp. 67–75). ACM Press, New York, NY.

MATHEUS, C. & RENDELL, L. [1989]. "Constructive induction on decision trees." In *Proc. Eleventh International Joint Conference on Artificial Intelligence* (pp. 645–50). Morgan Kaufmann, San Mateo, CA.

MEISEL, W. & MICHALOPOULOS, D. [1973]. "A partitioning algorithm with application in pattern classification and the optimization of decision trees." *IEEE Trans. Comput.* **22**, 93–103.

MERCKT, T. VAN DE [1993]. "Decision trees in numerical attribute spaces." In *Proc. Thirteenth International Joint Conference on Artificial Intelligence* (pp. 1016–21). Morgan Kaufmann, San Mateo, CA.

MICHALSKI, R. [1983]. "A theory and methodology of inductive learning." In R. Michalski, J. Carbonell & T. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach* (pp. 83–134). Tioga, Palo Alto, CA.

MICHALSKI, R. & CHILAUSKY, R. [1980]. "Knowledge acquisition by encoding expert rules versus computer induction from examples: A case study involving soybean pathology." *Int. J. Man–Mach. Stud.* **12**, 63–87.

MICHALSKI, R., MOZETIČ, I., HONG, J. & LAVRAČ, N. [1986]. "The multi-purpose incremental learning system AQ15 and its testing application to three medical domains." In *Proc. Fifth National Conference on Artificial Intelligence* (pp. 1041–5). Morgan Kaufmann, Los Altos, CA.

MICHIE, D. [1986]. *On Machine Intelligence*. Ellis Horwood, London.

MICHIE, D., SPIEGELHALTER, D. & TAYLOR, C. (EDS.) [1994]. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, London.

MINGERS, J. [1989a]. "An empirical comparison of selection measures for decision-tree induction." *Mach. Learn.* **3**, 319–42.

——— [1989b]. "An empirical comparison of pruning methods for decision tree induction." *Mach. Learn.* **4**, 227–43.

MINSKY, M. & PAPERT, S. [1969]. *Perceptrons*. MIT Press, Cambridge, MA.

MORET, B. [1982]. "Decision trees and diagrams." *ACM Comput. Surv.* **14**, 593–623.

MURPHY, O. & MCCRAW, R. [1991]. "Designing storage efficient decision trees." *IEEE Trans. Comput.* **40**, 315–20.

MURPHY, P. & AHA, D. [1994]. UCI repository of machine learning databases (http://www.ics.uci.edu/~mlearn/MLRepository.html). Department of Information and Computer Science, University of California at Irvine.

MURTHY, S., KASIF, S. & SALZBERG, S. [1994]. "A system for induction of oblique decision trees." *J. Artif. Intell. Res.* **2**, 1–32.

NATARAJAN, B. [1991]. "Probably approximate learning of sets and functions." *SIAM J. Comput.* **20**, 328–51.

NIBLETT, T. & BRATKO, I. [1986]. "Learning decision rules in noisy domains." In M. Bramer (ed.), *Proc. Research and Development in Expert Systems III* (pp. 25–34). Cambridge University Press, Cambridge.

NOORDEWIER, M., TOWELL, G. & SHAVLIK, J. [1991]. "Training knowledge-based neural networks to recognize genes in DNA sequences." In R. Lippmann, J. Moody & D. Touretzky (eds.), *Advances in Neural Information Processing Systems* **3** (pp. 530–6). Morgan Kaufmann, San Mateo, CA.

NÚÑEZ, M. [1988]. "Economic induction: A case study." In D. Sleeman (ed.), *Proc. Third European Working Session on Learning* (pp. 139–45). Pitman, London.

OLIVER, J. [1993]. "Decision graphs—an extension of decision trees." In *Preliminary papers of the Fourth International Workshop on Artificial Intelligence and Statistics* (pp. 343–50). Society for Artificial Intelligence and Statistics.

PAGALLO, G. & HAUSSLER, D. [1990]. "Boolean feature discovery in empirical learning." *Mach. Learn.* **5**, 71–99.

PAYNE, H. & MEISEL, W. [1977]. "An algorithm for constructing optimal binary decision trees." *IEEE Trans. Comput.* **26**, 905–16.

PIATETSKY-SHAPIRO, G. (ED.) [1991]. *Knowledge Discovery in Databases*. MIT Press, Cambridge, MA.

QUINLAN, R. [1983]. "Learning efficient classification procedures and their application to chess end-games." In R. Michalski, J. Carbonell & T. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach* (pp. 391–411). Tioga, Palo Alto, CA.

——— [1986a]. "The effect of noise in concept learning." In R. Michalski, J. Carbonell & T. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach II* (pp. 149–66). Morgan Kaufmann, Los Altos, CA.

——— [1986b]. "Induction of decision trees." *Mach. Learn.* **1**, 81–106.

——— [1987a]. "Decision trees as probabilistic classifiers." In *Proc. Fourth International Workshop on Machine Learning* (pp. 31–7). Morgan Kaufmann, Los Altos, CA.

——— [1987b]. "Generating production rules from decision trees." In *Proc. Tenth International Joint Conference on Artificial Intelligence* (pp. 304–7). Morgan Kaufmann, Los Altos, CA.

——— [1987c]. "Simplifying decision trees." *Int. J. Man–Mach. Stud.* **27**, 221–34.

——— [1988a]. "An empirical comparison of genetic and decision-tree classifiers." In J. Laird (ed.), *Proc. Fifth International Conference on Machine Learning* (pp. 135–41). Morgan Kaufmann, San Mateo, CA.

——— [1988b]. "Decision trees and multi-valued attributes." In J. Hayes, D. Michie & J. Richards (eds.), *Machine Intelligence 11: Logic and the Acquisition of Knowledge* (pp. 305–18). Oxford University Press, Oxford.

——— [1989]. "Unknown attribute values in induction." In A. Segre (ed.), *Proc. Sixth International Workshop on Machine Learning* (pp. 164–8). Morgan Kaufmann, San Mateo, CA.

——— [1990a]. "Decision trees and decisionmaking." *IEEE Trans. Syst. Man Cybern.* **20**, 339–46.

——— [1990b]. "Learning logical definitions from relations." *Mach. Learn.* **5**, 239–66.

——— [1990c]. "Probabilistic decision trees." In Y. Kodratoff & R. Michalski (eds.), *Machine Learning: An Artificial Intelligence Approach III* (pp. 140–52). Morgan Kaufmann, San Mateo, CA.

——— [1991]. "Improved estimates for the accuracy of small disjuncts." *Mach. Learn.* **6**, 93–8.

——— [1993]. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA.

——— [1996]. "Improved use of continuous attributes in C4.5." *J. Artif. Intell. Res.* **4**, 77–90.

QUINLAN, R. & RIVEST, R. [1989]. "Inferring decision trees using the minimum description length principle." *Inf. Comput.* **80**, 227–48.

RAEDT, L. DE & BRUYNOOGHE, M. [1992]. "A unifying framework for concept-learning algorithms." *The Knowl. Eng. Rev.* **7**, 251–69.

RISSANEN, J [1989]. *Stochastic Complexity in Statistical Inquiry*. World Scientific, River Edge, NJ.

——— [1995]. "Stochastic complexity in learning." In P. Vitányi (ed.), *Proc. Second European Conference on Computational Learning Theory*, Lecture Notes in Computer Science **904** (pp. 196–210). Springer-Verlag, Berlin.

RIVEST, R. [1987]. "Learning decision lists." *Mach. Learn.* **2**, 229–46.

ROUSU, J. [1996]. Constructing decision trees and lists using the MDL principle. M.Sc. thesis, Rep. C-1996-15. Department of Computer Science, University of Helsinki. (In Finnish).

RUMELHART, D. & MCCLELLAND, J. (EDS.) [1986]. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1: Foundations*. MIT Press, Cambridge, MA.

SAFAVIAN, R. & LANDGREBE, D. [1991]. "A survey of decision tree classifier methodology." *IEEE Trans. Syst. Man Cybern.* **21**, 660–74.

SAITTA, L. & BERGADANO, F. [1993]. "Pattern recognition and Valiant's learning framework." *IEEE Trans. Pattern Anal. Mach. Intell.* **15**, 145–55.

SAKAKIBARA, Y. [1991]. "On learning from queries and counterexamples in the presence of noise." *Inf. Process. Lett.* **37**, 279–84.

——— [1993]. "Noise-tolerant Occam algorithms and their applications to learning decision trees." *Mach. Learn.* **11**, 37–62.

SHACKELFORD, G. & VOLPER, D. [1988]. "Learning k-DNF with noise in the attributes." In D. Haussler & L. Pitt (eds.), *Proc. 1988 Workshop on Computational Learning Theory* (pp. 97–103). Morgan Kaufmann, San Mateo, CA.

SHAPIRO, A. [1983]. The role of structured induction in expert systems. Ph. D. Thesis. University of Edinburgh.

SHAVLIK, J., MOONEY, R. & TOWELL, G. [1991]. "Symbolic and neural learning algorithms: An experimental comparison." *Mach. Learn.* **6**, 111–43.

SHEPHERD, B. [1983]. "An appraisal of a decision tree approach to image classification." In A. Bundy (ed.), *Proc. Eighth International Joint Conference on Artificial Intelligence* (pp. 473–5). William Kaufmann, Los Altos, CA.

SHEPHERD, B., PIPER, J. & RUTOVITZ, D. [1988]. "Comparison of ACLS and classical linear methods in a biological application." In J. Hayes, D. Michie & J. Richards (eds.), *Machine Intelligence 11: Logic and the Acquisition of Knowledge* (pp. 423–34). Oxford University Press, Oxford.

SILLITOE, I. & ELOMAA, T. [1994]. "Learning decision trees for mapping the local environment in mobile robot navigation." In *Proc. SMC-COLT Workshop on Robot Learning* (pp. 119–25).

SIMON, H. [1991]. "The Vapnik-Chervonenkis dimension of decision trees with bounded rank." *Inf. Process. Lett.* **39**, 137–41.

——— [1995]. "Learning decision lists and trees with equivalence-queries." In P. Vitányi (ed.), *Proc. Second European Conference on Computational Learning Theory*, Lecture Notes in Computer Science **904** (pp. 322–36). Springer-Verlag, Berlin.

SLEEMAN, D. [1994]. "Towards a technology and a science of machine learning." *AI Comm.* **7**, 29–38.

SLOAN, R. [1988]. "Types of noise in data for concept learning." In D. Haussler & L. Pitt (eds.), *Proc. 1988 Workshop on Computational Learning Theory* (pp. 91–6). Morgan Kaufmann, San Mateo, CA.

TAN, M. [1993]. "Cost-sensitive learning of classification knowledge and its application in robotics." *Mach. Learn.* **13**, 7–33.

TAN, M. & SCHLIMMER, J. [1990]. "Two case studies in cost-sensitive concept acquisition." In *Proc. Eighth National Conference on Artificial Intelligence* (pp. 854–60). MIT Press, Cambridge, MA.

TSATSARAKIS, C. & SLEEMAN, D. [1993]. "Supporting preprocessing and postprocessing for machine learning algorithms: A workbench for ID3." *Knowl. Acquisition* **5**, 367–84.

TURING, A. [1950]. "Computing machinery and intelligence." *Mind* **LIX**, 433–60.

UTGOFF, P. [1986]. *Machine Learning of Inductive Bias.* Kluwer, Norwell, MA.

——— [1989]. "Incremental induction of decision trees." *Mach. Learn.* **4**, 161–86.

——— [1994]. "An improved algorithm for incremental induction of decision trees." In W. Cohen & H. Hirsh (eds.), *Machine Learning: Proc. Eleventh International Conference* (pp. 318–25). Morgan Kaufmann, San Francisco, CA.

——— [1995]. Decision tree induction based on efficient tree restructuring. Rep. 95-18. Department of Computer Science, University of Massachusetts.

VALIANT, L. [1984]. "A theory of the learnable." *Commun. ACM* **27**, 1134–42.

——— [1985]. "Learning disjunctions of conjunctions." In *Proc. Ninth International Joint Conference on Artificial Intelligence* (pp. 560–6). Morgan Kaufmann, Los Altos, CA.

VELDE, W. VAN DE [1990]. "Incremental induction of topologically minimal trees." In B. Porter & R. Mooney (eds.), *Proc. Seventh International Conference on Machine Learning* (pp. 66–74). Morgan Kaufmann, San Mateo, CA.

WALLACE, C. & PATRICK, J. [1993]. "Coding decision trees." *Mach. Learn.* **11**, 7–22.

WEISS, S. & KAPOULEAS, I. [1989]. "An empirical comparison of pattern recognition, neural nets, and machine learning classification methods." In *Proc. Eleventh International Joint Conference on Artificial Intelligence* (pp. 781–7). Morgan Kaufmann, San Mateo, CA.

WIENER, N. [1948]. *Cybernetics*. John Wiley & Sons, New York.

Appendix A

Proofs of Lemmas and Theorems

This appendix gives rigorous proofs for those lemmas and theorems for which full proofs were omitted in the text. All important proofs are included in the text; those presented here are mostly simple variations of proofs presented earlier in the text or elsewhere. We proceed in the order of the text chapters.

Chapter 2

First, Theorem 6 is proved by modifying the corresponding proof of Sakakibara [1993] only slightly. For the proof we need Hoeffding's [1963] inequalities (Lemma 21) and the following lemma.

Lemma 25 *Let* $\mathbf{F}$ *be a polynomial-sized function class and let* η_b *be such that* $\eta \leq \eta_b \leq \eta + \varepsilon(1 - 2\eta)/2$. *Let* $Occam(S, \varepsilon, \delta, \eta_b)$ *be a noise-tolerant Occam algorithm for* $\mathbf{F}$. *When given a sample* S *of* m *examples drawn from the oracle* $EX_\eta(P, f)$ *the algorithm outputs a hypothesis* g *such that* $P(f \triangle g) \leq \varepsilon$ *with probability at least* $1 - \delta$. *The sample size* m *required is at least*

$$\frac{8}{\varepsilon^2(1/2 - \eta_b)^2} \ln \frac{2|F_{n,k}|}{\delta}.$$

Proof Let us consider the disagreements of the hypothesis g and the sample S; i.e., examples $\langle x, l \rangle \in S$ such that $g(x) \neq l$. The probability that an example $\langle x, l \rangle$ drawn from $EX_\eta(P, f)$ disagrees with g is

- the probability that $x \in \{\, y \in [m]^n \mid f(y) \neq g(y) \,\}$ and l is not corrupted by the oracle, which is $P(f \triangle g) \cdot (1 - \eta)$, plus

- the probability that $x \in \{ y \in [m]^n \mid f(y) = g(y) \}$ and l is corrupted by the oracle, which is $(1 - P(f \triangle g))\eta$.

Summing these together, we have that the probability, that an example in S disagrees with the hypothesis g, is

$$P(f \triangle g) \cdot (1-\eta) + (1 - P(f \triangle g))\eta = \eta + P(f \triangle g) \cdot (1 - 2\eta).$$

For the target function f we have $D(f, S) = \eta|S|$ and for any function g, such that $P(f \triangle g) = \varepsilon$, we have $D(g, S) = (\eta + \varepsilon(1-2\eta))|S|$. Thus, any such g has an expected rate of disagreement that is greater by at least $s = \varepsilon(1 - 2\eta)$ than that of the target function.

By the assumption $\eta \leq \eta_b$ and the first Hoeffding's inequality, the probability that the target function f has more than $(\eta_b + s/4)m$ disagreements with a sample S of m examples drawn from $EX_\eta(P, f)$ is

$$\begin{aligned} GE(\eta, m, \eta_b + s/4) &\leq GE(\eta_b, m, \eta_b + s/4) \\ &\leq e^{-2(s/4)^2 m} \\ &\leq e^{-2(1/16)8 \ln \frac{2|\mathbf{F}|}{\delta}} \\ &\leq e^{-\ln \frac{2|\mathbf{F}|}{\delta}} \\ &= \frac{\delta}{2|\mathbf{F}|}. \end{aligned}$$

Hence, with probability at least $1 - \delta/2$, *Occam*$(S, \varepsilon, \delta, \eta_b)$ can find and output a function $g \in \mathbf{F}$, such that

$$\frac{D(g, S)}{m} \leq \eta_b + \frac{s}{4}.$$

The probability that a function with error greater than ε has at most $(\eta_b + s/4)m$ disagreements is, by the assumption $\eta_b \leq \eta + \varepsilon(1 - 2\eta)/2 = \eta + s/2$ and by using the second Hoeffding's inequality, at most

$$\begin{aligned} LE(\eta + \varepsilon(1-2\eta), m, \eta_b + s/4) &\leq LE(\eta_b + s/2, m, \eta_b + s/4) \\ &\leq e^{-2(s/4)^2 m} \\ &= \frac{\delta}{2|\mathbf{F}|}. \end{aligned}$$

Since there are at most $|\mathbf{F}|$ functions in $\mathbf{F}$, the probability of producing a function with error greater than ε is less than

$$|\mathbf{F}| \cdot e^{-2(s/4)^2 m} = \delta/2.$$

Therefore, with probability at least $1 - \delta$, *Occam*$(S, \varepsilon, \delta, \eta_b)$ outputs a function g such that $P(f \triangle g) \leq \varepsilon$. □

Algorithm A.1 *OccamLearn*$(S, \varepsilon, \delta, \eta_b)$
input: a nonempty noisy sample S of some n-ary function on $[m]$, and positive reals η_b, ε, and δ, such that $\eta \leq \eta_b < 1/2$ and $0 < \varepsilon, \delta \leq 1$.
output: a function g.
begin
(1) $\eta_e \leftarrow \eta_b$;
(2) $Q \leftarrow \emptyset$;
(3) **while** $\eta_e > 0$ **do**
(4) $\quad Q \leftarrow Q \cup Occam(S, \varepsilon, \delta, \eta_e)$;
(5) $\quad \eta_e \leftarrow \eta_e - \varepsilon(1 - 2\eta_b)/2$
od;
(6) **return** a function $g \in Q$ such that it minimizes $D(g, S)$, the number of disagreements with the sample S
end

Now we can prove the actual theorem.

Theorem 6 *Let* **F** *be a polynomial-sized function class and let* η_b *be such that* $\eta \leq \eta_b \leq \eta + \varepsilon(1 - 2\eta)/2$. *If there exists a noise-tolerant Occam algorithm for* **F**, *then* **F** *is polynomially learnable in the presence of classification noise. The sample size required is at least*

$$\frac{8}{\varepsilon^2(1/2 - \eta_b)^2} \ln \frac{2|F_{n,k}|}{\delta}.$$

Proof Let *Occam*$(S, \varepsilon, \delta, \eta_b)$ be the noise-tolerant Occam algorithm for **F**. Using the algorithm *Occam*, we can construct a learning algorithm *OccamLearn* for **F** that, with probability $1-\delta$, outputs a hypothesis g, such that $P(f \bigtriangleup g) \leq \varepsilon$, for the target function f from the oracle $EX_\eta(P, f)$, where $f \in F_{n,k}$ and P is an arbitrary probability distribution on **F**.

In the sequence of successively smaller values η_e examined by *OccamLearn* there will be one such that $\eta \leq \eta_e \leq \eta + \varepsilon(1 - 2\eta)/2$ because η_e is decreasing by $\varepsilon(1 - 2\eta_b)/2$ and, by assumption, $\varepsilon(1 - 2\eta_b) \leq \varepsilon(1 - 2\eta)$. Then, because of Lemma 25, the lower bound on m implies that, with probability at least $1 - \delta/2$, *OccamLearn* will produce at least one function $g \in Q$ such that

$$\begin{aligned} \frac{D(g, S)}{m} &\leq \eta_e + \varepsilon(1 - 2\eta_e)/4 \\ &\leq \eta + \frac{3}{4}\varepsilon(1 - 2\eta). \end{aligned}$$

The probability that a function $h \in \mathbf{F}$ with error greater than ε has $D(h,S)/m \leq \eta + 3/4 \cdot \varepsilon(1-2\eta)$ is at most

$$LE\left(\eta + \varepsilon(1-2\eta), m, \eta + \frac{3}{4}\varepsilon(1-2\eta)\right) \leq e^{-2(\varepsilon(1-2\eta)/4)^2 m}.$$

Since there are $|\mathbf{F}|$ functions in $\mathbf{F}$, the probability of producing a function with error greater than ε is less than

$$|\mathbf{F}| \cdot e^{-2(\varepsilon(1-2\eta)/4)^2 m} < \delta/2.$$

Hence with probability at least $1-\delta$ *OccamLearn* outputs a function g such that $P(f \triangle g) \leq \varepsilon$.

□

Chapter 3

Detailed proof for the time requirement of *IFM* (Theorem 16) was not given in the text. We prove the result in the following by a verbal argumentation, rather than giving a rigorous mathematical proof, which would require long and obscure presentation. The following argumentation—together with the empirical evidence—ought to suffice for our current needs. The following lemma simplifies the proof of the main result substantially.

Lemma 26 *Let $F_{FM}(S)$ denote the time spent by Findmin, in the worst case, on the calls to Find when given the sample S as input. Respectively, let $F_{IFM}(S)$ denote the time spent by IFM, in the worst case, on the calls to Find when processing the sequence S of examples. Then, $F_{IFM}(S) \leq F_{FM}(S)$.*

Proof First, observe that, due to the incremental processing of examples, a variable can only change from uninformative to informative. In other words, the addition of an instance to the sample can change some uninformative attributes into informative ones, but never the other way round. Therefore, any attribute that is informative (on a subset) from the outset will remain such for (a subset of) the full sample. Because of this, in *IFM* variables are never needlessly examined; i.e., every attribute that is attempted, really is an informative one w.r.t. (a subset of) the final sample S, and ought to have been examined.

Now, consider the calls to *Find* made by *Findmin* for a sample S of rank r. First, it must determine that the sample does not have rank $0, \ldots, r-1$ by ascertaining that none of the trees with lower rank than r is consistent with the sample. In the worst case that involves examining all permutations of informative variables for all rank candidates. Only if an exhausted node appears in the evolving tree, does *Findmin* avoid examining some candidate trees. However, in the worst-case time consumption of *Findmin* (Theorem 11) this optimization cannot be taken into account. Thereafter, *Findmin* still has to come up with the tree of correct rank by continuing the recursive search procedure.

It is easy to see that at most the same time as spent by *Findmin* on the calls to *Find* is required by *IFM*, since it does not waste any effort by examining false variables. All calls to *Find* that are made by *IFM* also have been made, in the worst case, during *Findmin*'s execution. The bookkeeping in the incremental algorithm ensures that an attribute is never examined more than once in a given permutation of variables in the evolving tree. Thus, the time spent by *IFM* on the calls to *Find*, in the worst case, cannot exceed that of *Findmin*. Therefore, $F_{IFM}(S) \leq F_{FM}(S)$. $\square$

Theorem 16 *Given a sample S of a n-ary function on $[m]$, using IFM(S) we can produce a decision tree that is consistent with S and has rank $r(S)$ in time $O(|S|m^{r(S)}(n+1)^{2r(S)})$.*

Proof It is easy to see that the additional bookkeeping in *IFM* does not raise its asymptotic time requirement beyond that of *Findmin*. In *IFM* every example in the sequence S has to be directed down the tree, which, in the worst case, requires examining the value of the i informative variables. Only constant time operations are performed for the examples. Thus, the bookkeeping may require, in the worst case, time $O(|S|i) \leq O(|S|n)$, which is dominated by the time required for calls to *Find* and, therefore, does not affect the algorithm's asymptotic time requirement. The remaining operations (stack management and subtree substitutions) are also constant time operations and, therefore, do not change the asymptotic time requirement. Hence, because of Lemma 26, the claim follows. $\square$

Appendix B

Description of the Test Domains

This appendix presents the test domains used in the empirical experiments in Chapter 5 in more detail than has been applied earlier. For each data set we give a general description of the real-world domain it represents, if any, the attributes that are used to comprise the real-world information, and the classification task at hand. The past usage of each data set and the special characteristics reported in the literature are also reviewed.

B.1 Assessing credit card applications

Background This is a database containing confidential information taken from credit card applications. Therefore, all attribute names and values have been changed to meaningless symbols to protect confidentiality of the data. Nevertheless, the domain concerns approval of credit card applications on the basis of simple financial facts.

Attributes and examples There are 14 attributes, 8 of which are categorical and 6 continuous. The discretized version of the data offered to *Rank* has on average 6.1 values per attribute. There are 690 examples in the domain.

Noise and missing values In this version of the data set, which comes from the StatLog group [Michie *et al*. 1994], there are no missing values. Originally there were a few missing values, but they were replaced by the overall median (mode of the attribute for a nominal attribute and mean of the attribute for a numerical attribute).

Class distribution Of the total 690 examples 307 (44.5%) belong to category approved and the remaining 383 (55.5%) examples are disapproved applications.

B.2 Vehicle type identification

Background Image recognition is often thought to constitute an unsurmountable obstacle to empirical learning algorithms. Neural networks are usually offered for the job. Unfortunately in an autonomous mobile robot, for example, one does not have the luxury of iterating the learning period indefinitely until the network converges [Shavlik *et al*. 1991], but one has to be able to make prompt decisions. Therefore, empirical learners, too, have to be considered for these tasks [Tan 1993, Sillitoe & Elomaa 1994].

This data was originally gathered at the Turing Institute [Michie *et al*. 1994]. Images of four model vehicles were used: A double decker bus, Chevrolet van, Saab 9000, and Opel Manta. It was anticipated to be easy to distinguish the bus and the van from the cars, but that it would be harder to make a distinction between the two cars. 18 features were extracted from the 128 $\times$ 128 grey scale pictures.

Attributes and examples The 18 features attempt to characterize shape of the object in the image. They include typical attributes in wave-based recognition, for instance: Circularity, radius ratio, compactness, scaled variance along major and minor axes, etc. After categorization, the attributes have on average 7 values in their range. There are 846 examples in this data set.

Noise and missing values There is no explicit noise in the data, only that which is inherent due to the measuring apparatus. All values are known.

Class distribution The images are distributed pretty evenly between the different vehicles: There are 218 (25.8%) examples of buses, 199 (23.5%) vans, 217 (25.7%) Saabs, and 212 (25.1%) Opels in the data.

B.3 Diabetes prediction

Background Application of machine learning techniques, and artificial intelligence techniques more generally, to medical domains has been studied a lot over the years. The main reason for this being the availability of suitable data and proper classification tasks. Furthermore, medical decision making is in part so clearly based on heuristic methods that parallels with artificial intelligence techniques are easy to draw.

The data in hand comes from the National Institute of Diabetes and Digestive and Kidney Diseases. The diagnostic, binary-valued variable investigated is whether the patient shows signs of diabetes according to World Health Organization criteria; i.e., if the two-hour post-load plasma glucose was at least 200 mg/dl at any survey examination or if found during routine medical care. The population lives near Phoenix, Arizona, USA. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Attributes and examples The attributes constitute of eight medical measurements taken at a routine examination. They are:

number of times pregnant,	
plasma glucose concentration	(a 2 hours in an oral glucose tolerance test),
diastolic blood pressure	(mm Hg),
triceps skin fold thickness	(mm),
two-hour serum insulin	(μU/ml),
body mass index	(weight in kg/(height in m)^2),
diabetes pedigree function, and	
age	(years).

Discretization yields, on average, 12.5 values for each attribute. The domain consists of 768 examples.

Noise and missing values There is no explicit noise present nor any recordings missing.

Class distribution There are 500 (65.1%) patients that have tested positive and 268 (34.9%) ones that tested negative.

B.4 Space shuttle radiator positioning

Background This is a large database concerning actual NASA Space Shuttle problem: The position of space radiators in the shuttle [Catlett 1991]. The seven classes represent the possible states of the radiators. Data has been gathered during two separate flights. Very small prediction error (far below 1%) is obtained by a decision tree of seven nodes [Michie *et al*. 1994]. The original data contains hundreds of thousands of examples, but in the StatLog version of the domain there are 58,000 examples.

Attributes and examples There are only 9 attributes describing instances. They comprise measurements of three sensors that are monitored at one second intervals. All attributes were originally numerical, but in our tests we have also used a discretized version of the data (see Chapter 5 for a more thorough account of this). In the discretized version attributes have on average 2.9 values in their range. The total number of examples in this domain is 58,000. It has been divided into a training set of 43,500 examples and a test set of 14,500 examples.

Noise and missing values The data appears to be noise free, since arbitrarily small error rates can be attained given sufficient data. There are no missing values in the data.

Class distribution Approximately 80% of the data belongs to class “radiator flow”. The class distribution is relatively skewed otherwise also: For instance, class “bpv close” has only 10 instances, which do not count as even a per mill of the full data. The following table gives the exact numbers of examples per class.

CLASS DISTRIBUTION				
	FULL DATA		TEST DATA	
`rad flow`	45,586	(78.6%)	11,478	(79.2%)
`fpv close`	50	(0.1%)	13	(0.1%)
`fpv open`	171	(0.3%)	39	(0.3%)
`high`	8,903	(15.4%)	2,155	(14.9%)
`bypass`	3,267	(5.6%)	809	(5.6%)
`bpv close`	10	(0.0%)	4	(0.0%)
`bpv open`	13	(0.0%)	2	(0.0%)
TOTAL	58,000		14,500	

B.5 DNA sequence boundaries

Background Biological data has gained importance recently. In particular, the massive exploration into the human genome has brought much attention to these applications of machine learning. Especially, DNA manipulation and interpretation is of prime interest.

Splice junctions are points on a DNA sequence at which "superfluous" DNA is removed during the process of protein creation in higher organisms. The problem posed in this data set is to recognize, given a sequence of DNA, the boundaries between *exons*—the parts of the DNA sequence retained after splicing—and *introns*—the parts that are spliced out.

Here the primary decision to make is whether the center point in the DNA sequence window presented is a splice junction or not. This problem consists of two subtasks: Recognizing exon/intron boundaries (referred to as EI sites), and recognizing intron/exon boundaries (IE sites). In the biological community, IE borders are referred to as *acceptors* while EI borders are referred to as *donors*.

Attributes and examples The StaLog DNA data set [Michie *et al*. 1994] is a processed version of the University of California at Irvine repository data set [Murphy & Aha 1994]. The main difference is that the symbolic variables representing the nucleotides (only `A`, `G`, `T`, `C`) were replaced by 3 binary indicator variables. Thus the original 60 symbolic attributes were changed into 180 binary attributes. The names of the examples were removed. The examples with ambiguities were removed (there was only four of them). According to the suggestion of the StatLog group, we chose to use a further processed version of this data: 120 more or less irrelevant attributes were deleted, and only the remaining 60 binary attributes were utilized. Training set consists of 2,000 examples and the test set contains 1,186 examples.

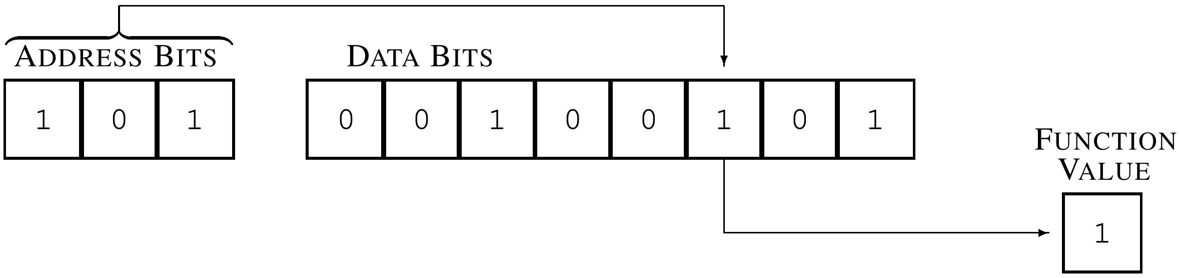
Noise and missing values The examples are not affected by noise and neither are any values missing from the data.

Class distribution The examples in the training and test sets are divided into the three classes as described in the following table.

CLASS DISTRIBUTION				
	TRAINING DATA		TEST DATA	
`ei`	464	(23.2%)	303	(25.6%)
`ie`	485	(24.3%)	280	(23.6%)
`neither`	1,051	(52.6%)	603	(50.8%)
TOTAL	2,000		1,186	

B.6 The six-bit multiplexor function

Background The family of multiplexor functions contains, for each positive integer k, a Boolean function defined on $k + 2^k$ attributes, or *bits*. A multiplexor function is a simple Boolean function that can be considered consisting of k *address bits* and 2^k *data bits*. The k address bits are capable of indexing the space of 2^k data bits. The value of the multiplexor function is defined to be the value of the data bit determined by the values of the k address bits together (see the picture below).



The domain in hand is the six-bit multiplexor function ($k = 2$), which has two address bits and four data bits. In addition, five further bits that are neither address nor data bits are used to define the function. The latter five bits are called *irrelevant bits*, since their values have no effect on the outcome of the multiplexor function. An irrelevant bit has both Boolean values in the sample for each configuration of the other bits' values.

Ever since Quinlan [1988a] showed that the straightforward top-down approach of decision tree learning is inherently incapable of learning the most natural (and minimal at the same time) tree representations for these functions, the multiplexor functions have appeared repeatedly in machine learning literature (e.g., [Pagallo & Haussler 1990, Utgoff 1989, Van de Velde 1990]). In fact, one could say that multiplexor functions have been adopted as one of the first simple standard test cases to manage by all new learning techniques. Overcoming problems related to multiplexor functions have even inspired novel techniques [Van de Velde 1990].

Attributes and examples Each attribute in this domain is a Boolean one. Thus they all have two values: **true** and **false**. The address bits are called `Addr0` and `Addr1`, data bits are named `Data0`,..., `Data3`, and the irrelevant bits are named `Irrel1`,..., `Irrel5`.

Noise and missing values There are no missing value recordings for any of the ten attributes or the class information among the 2048 instances of the sample.

Class distribution The examples, naturally, are divided evenly among the two classes.

B.7 LED digit identification

Background Simple displays put together from light-emitting diodes (LEDs) are common, for instance, in modern-day household appliances and similar apparatus. The diodes in such displays have two states: They are either **on** (lit) or **off** (put out). Thus the LEDs can be modeled by Boolean variables.

The domain in hand concerns the identification of digits in an ordinary pocket calculator display. The display has 7 LEDs per character (see the picture above). The classes are the ten decimal digits. However, the calculator is a broken one and gives faulty images at times: Each LED has a 10% error rate. This noisy domain of seven-LED display digit categorization was used as an example in the seminal book on decision tree learning by Breiman *et al*. [1984], and the domain has been one of the standard initial testing grounds for learning tools ever since (e.g., [Buntine & Niblett 1992, Quinlan 1987b, Quinlan & Rivest 1989]).

Attributes and examples The data consists of 200 such noisy images and their correct interpretations. Each example is described by 7 Boolean attributes.

Noise and missing values A 10% noise rate affects the attributes, the classification of an instance is noise-free. There are no values missing from this data.

Class distribution The class distribution of the 200 examples is described in the following table.

CLASS DISTRIBUTION					
1	18	(9.0%)	6	28	(14.0%)
2	24	(12.0%)	7	19	(9.5%)
3	20	(10.0%)	8	22	(11.0%)
4	12	(6.0%)	9	24	(12.0%)
5	13	(6.5%)	0	20	(10.0%)
TOTAL				200	

B.8 Chess endgame result

Background The game of chess and other board games have traditionally attracted attention as a testing ground for artificial intelligence ideas. These games are considered to require enough mental skills in a restricted setting to give reliable evidence of the techniques' capabilities [Michie 1986]. Inasmuch as chess is considered, it is an extremely complicated game difficult even for humans to master. Artificial intelligence approaches to chess have had to create specialized hardware to narrow the gap to human performance. When simpler and easier managed chess domains are desired, the number of pieces can be reduced. Thus, one comes around to consider chess endgames, where all but some pieces have already been captured.

The domain CHESS concerns endgames where, in addition to the King, White has a Pawn on square a7 and Black has a Rook. All other pieces, except the Pawn, may be situated anywhere on the board. The next move is to be made by White. The decision classes are "won" and "not-won" for White. Even this seemingly simple domain has proven to be too complex to be solved by conventional programming techniques [Shapiro 1983].

This particular chess domain [Shapiro 1983] and other similar endgame domains [Michie 1986, Quinlan 1986a, Quinlan 1986b] have been used extensively to test the performance of symbolic learning techniques throughout the years.

Attributes and examples The 36 attributes are result of a detailed analysis of the endgame in hand, and they all comprise an answer to a relevant question like "Does one or more Black pieces control the queening square?" or "Is there a potential skewer as opposed to fork?". Hence, much more chess knowledge than just the basic board configuration is encoded into the attribute values. A detailed analysis of the task and complete account of the attributes is given by Shapiro [1983, Chapter 7].

The entire set of 3,196 instances results from the 209,718 legal King and Pawn on a7 versus King and Rook positions with White to move [Shapiro 1983].

Noise and missing values There are no missing values in the data nor noise prevailing.

Class distribution There are 1,669 examples (50.2%) in the class won and the rest (1,527 examples, 49.8%) belong to class "not-won".

B.9 Primary tumor location

Background The second medical domain concerns cancer. This time the focus is on predicting the location of a primary tumor. Physicians distinguish between 22 possible locations, which are the classes of this domain. The location of primary tumor is one of the important sources of evidence used in selecting cancer treatment.

The instances of this domain have been shown to 4 internists (non-specialists) and to 4 oncologists (specialists) at the Institute of Oncology, University Medical Center in Ljubljana, Slovenia, where the data was originally compiled. The internists and oncologists were asked to classify the instances, and their accuracy was checked. They obtained 32% and 42% correct classification rate, respectively [Michalski *et al*. 1986].

Attributes and examples The description data here is straightforward: 13 of the total 17 attributes answer whether metastases have been detected at a given location (liver and brain, for example). The remaining attributes give, e.g., the age and the sex of the patient and the histologic type of carsinoma. The data set is inconsistent; i.e., there are examples with identical attribute values, but different classification. Since the data has been verified after collection (by operation or X-ray), it must be that the set of attributes is incomplete [Clark & Niblett 1989, Michalski *et al*. 1986]. The total number of examples is 339.

Noise and missing values Two attributes have several values missing: For attribute "histologic type" the value is not known in 67 examples and for "degree of diffe" in 155 examples. In addition, attributes "sex," "skin," and "axillar" lack the recording in one example.

Class distribution The following table makes the classification of examples explicit by listing the number of instances in the domain for each class. The relative portion of the examples per a class is also given.

CLASS DISTRIBUTION								
salivary glands	2	(0.6%)	gallblader	16	(4.7%)	ovary	29	(8.6%)
head & neck	20	(5.9%)	thyroid	14	(4.1%)	vagina	1	(0.3%)
esophasus	9	(2.7%)	kidney	24	(7.1%)	anus	0	(0.0%)
corpus uteri	6	(1.8%)	bladder	2	(0.6%)	colon	14	(4.1%)
stomach	39	(11.5%)	pancreas	28	(8.3%)	rectum	6	(1.8%)
duoden & sm.int	1	(0.3%)	prostate	10	(2.9%)	liver	7	(2.1%)
cervix uteri	2	(0.6%)	lung	84	(24.8%)	breast	24	(7.1%)
						testis	1	(0.3%)
TOTAL							339	

B.10 Soybean disease identification

Background The extremely famous data set of Michalski and Chilausky [1980] concerns the identification of diseases of soybeans in terms of macro-symptoms, which can be observed without sophisticated mechanical assistance. The intent being that a farmer, or even a layman should be able to make reliable observations [Michalski & Chilausky 1980].

Attributes and examples The 35 categorical attributes of this domain describe the growth environment (attributes 1–7), the plant in general (attributes 8–11), and local plant condition (attributes 12–35). The plant local descriptors are further refined to attributes describing the condition of leaves (12–18), stem (19–27), fruits or pods (28–29), seed (30–34), and root (35). The attributes are explained in more detail by Michalski and Chilausky [1980].

There are 15 classes; 4 categories have been eliminated from this task. The reason given is that the last four classes are unjustified by the data since they have so few examples.

Noise and missing values The following table lists those attributes that have missing value recordings and the number of such failings for each attribute.

ATTRIBUTE	MISSING	ATTRIBUTE	MISSING
2. `plant stand`	1	20. `lodging`	1
3. `precip`	8	21. `stem cankers`	41
4. `temp`	11	22. `canker lesion`	11
5. `hail`	7	23. `fruiting bodies`	11
6. `crop hist`	41	24. `external decay`	35
7. `area damaged`	1	25. `mycelium`	11
8. `severity`	1	26. `int discolor`	11
9. `seed tmt`	41	27. `sclerotia`	11
10. `germination`	41	28. `fruit pods`	11
11. `plant growth`	36	29. `fruit spots`	25
12. `leaves`	1	30. `seed`	35
14. `leafspots marg`	25	31. `mold growth`	29
15. `leafspot size`	25	32. `seed discolor`	29
16. `leaf shread`	25	33. `seed size`	35
17. `leaf malf`	26	34. `shriveling`	29
18. `leaf mild`	25	35. `roots`	35
19. `stem`	30		
		TOTAL	705

Class distribution The last table makes the classification of examples explicit by listing the number of instances in the domain for each class. The relative portion of the examples per a class is also given.

CLASS DISTRIBUTION					
diaporthe stem canker	10	(3.3%)	bacterial blight	10	(3.3%)
charcoal rot	10	(3.3%)	bacterial pustule	10	(3.3%)
rhizoctonia root rot	10	(3.3%)	purple seed stain	10	(3.3%)
phytophthora rot	40	(13.0%)	anthracnose	20	(6.5%)
brown stem rot	20	(6.5%)	phyllosticta leaf spot	10	(3.3%)
powdery mildew	10	(3.3%)	alternarialeaf spot	40	(13.0%)
downy mildew	10	(3.3%)	frog-eye leaf spot	40	(13.0%)
brown spot	40	(13.0%)			
TOTAL				290	

B.11 Mushroom species classification

Background This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota families belonging to the North American flora. There is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy. However, the following rule happens to hold in the random excerpt that is used in the experiments. It states that the edibility of the fungus can be determined by its odor and if it has no odor, then by the cap's color.

```
if odor = almond OR anise OR creosote
        THEN mushroom is Edible
if odor = fishy OR foul OR musty OR pungent OR spicy
        THEN mushroom is Poisonous
if the mushroom has no odor
        THEN
        if its cap has color yellow
                THEN mushroom is Poisonous
        OTHERWISE mushroom is Edible.
```

This domain was first used in Fisher's [1987] Ph.D. thesis and has ever since been used in several published studies (e.g., [Buntine & Niblett 1992, Holte 1993])

Attributes and examples The gillfungi are described in terms of their physical appearance, e.g., characters of the cap, the gill, and the stalk, their population type, and their habitats. Each agaric species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class has been combined with the poisonous one. We use a random excerpt of 2,065 examples of the full data, which contains over 8,000 examples.

Noise and missing values Only the attribute "stalk-root" has missing value recordings. Its value is not known for 935 instances.

Class distribution In the excerpt of 2,065 examples that is used in the experiments 1,375 (66.6%) examples are classified as edible and the remaining 690 (33.3%) examples are either definitely poisonous or of unknown edibility and have, thus, been classified as poisonous.

Appendix C

Dynamic Interface for Rank

This appendix lists the C shell script that acts as the dynamic interface between TELA and *Rank*. It has three parts: First the required variables are declared, then values communicated by TELA are assigned to them, and finally, the learning algorithm is evoked with the given parameters.

```
##############################################################
##                        VARIABLES                         ##
##############################################################
set echo
set callpath = '/home/fs/group/tela/bin/algorithms'
cd $callpath

set filename                # Attribute declaration file
set exsfilename             # Example vector file
set heuristics = 'no'       # Use Gini-index?
set incremental = 'no'      # Incremental mode?
set prevhypo
set initrank = 2            # Initial rank candidate
set level = 1
set kappa = 0
set gamma = 9000
set outfile
set mode                    # Induction or testing
set statsfile
set treefile
set accuracy
set rank
set size
```

```
##############################################################
##                  ASSIGN PARAMETER VALUES                 ##
##############################################################

while ($#argv)
      switch ($1)
             case 'STATSFILE=':
                  set statsfile = $2
                  shift
                  breaksw
             case 'STATS='
                  while ($#argv > 1)
                        shift
                        switch ($1)
                               case 'SIZE':
                                    set size = 'SIZE'
                                    breaksw
                               case 'RANK':
                                    set rank = 'RANK';
                                    breaksw
                               case 'ACCURACY':
                                    set accuracy = 'ACCURACY'
                                    breaksw
                               default:
                                    goto next_param
                        endsw
                  end
                  breaksw
             case 'ATTSFILE=':
                  set filename = $2
                  shift
                  breaksw
             case 'EXSFILE=':
                  set exsfilename = $2
                  shift
                  breaksw
             case 'CLASSIFIER=':
                  set treefile = $2
                  shift
                  breaksw
             case 'GINI'
                  set heuristics = 'yes'
                  breaksw
```

```
            case 'IFM':
                 set incremental = 'yes'
                 breaksw
            case 'PREVHYPO=':
                 set prevhypo = $2
                 shift
                 breaksw
            case 'INITRANK=':
                 set initrank = $2
                 shift
                 breaksw
            case 'LEVEL=':
                 set level = $2
                 shift
                 breaksw
            case 'KAPPA=':
                 set kappa = $2
                 shift
                 breaksw
            case 'GAMMA=':
                 set gamma = $2
                 shift
                 breaksw
            case 'MODE=':
                 if ($2 == 'TRAIN') then
                     set mode = 'i'
                 else
                     set mode = 'x'
                 endif
                 shift
                 breaksw
      endsw
      shift
      next_param:
end

\rm -f $callpath'/'rank.script
```

```
##############################################################
##        CALL ALGORITHM FOR INDUCTION OR TESTING           ##
##############################################################

if ($mode == 'x') then
        goto testing
endif

echo \"$filename\" > $callpath'/'rank.script
echo \"$exsfilename\" >> $callpath'/'rank.script
echo 'rank' >> $callpath'/'rank.script           # INDUCE !

echo $heuristics >> $callpath'/'rank.script
if ($heuristics == 'yes') then
        echo $level >> $callpath'/'rank.script
endif

echo $incremental >> $callpath'/'rank.script
if ($incremental == 'no') then
        echo $initrank >> $callpath'/'rank.script
endif

echo $kappa >> $callpath'/'rank.script
echo $gamma >> $callpath'/'rank.script
echo \"$statsfile\" >> $callpath'/'rank.script   # Statistics
echo \"$treefile\" >> $callpath'/'rank.script    # Resulting tree

$callpath'/'rank < $callpath'/'rank.script

exit 0

testing:

echo \"$filename\" > $callpath'/'rank.script
echo \"$exsfilename\" >> $callpath'/'rank.script
echo 'test' >> $callpath'/'rank.script           # TEST !
echo \"$treefile\" >> $callpath'/'rank.script    # Which tree?
echo \"$statsfile\" >> $callpath'/'rank.script   # Statistics

$callpath'/'rank < $callpath'/'rank.script

exit
```

Appendix D

Exact Measurements Under Different Noise Types

Table D.1: Exact values corresponding to the bar charts depicted in Fig. 5.2.

	ATTRIBUTE NOISE							
	20%				35%			
	TIME	RANK	SIZE	ACC.	TIME	RANK	SIZE	ACC.
Rank	9.8	**2.0**	140.9	94.6	7.7	**2.0**	94.9	85.8
C4.5	**2.2**	2.9	**63.8**	93.1	**2.7**	2.7	**68.2**	86.7
NewID	5.1	3.0	179.3	91.7	5.3	2.9	150.7	83.9
CN2	28.3	–	80.5	**95.8**	36.6	–	126.7	**90.8**
IFM	7.7	**2.0**	152.4	92.2	10.9	**2.0**	182.8	86.5
ITI	130.4	3.1	132.2	94.1	222.7	3.7	208.0	74.3

	CLASSIFICATION NOISE							
	20%				35%			
	TIME	RANK	SIZE	ACC.	TIME	RANK	SIZE	ACC.
Rank	4.5	1.1	20.2	89.3	5.3	**1.0**	**10.0**	**83.9**
C4.5	**3.6**	2.0	22.4	**91.2**	**4.3**	2.0	24.3	81.6
NewID	7.0	**1.0**	**10.0**	89.8	8.9	**1.0**	**10.0**	80.8
CN2	22.6	–	56.7	89.7	18.0	–	34.6	81.7
IFM	7.1	**1.0**	38.3	88.5	6.4	**1.0**	47.4	81.0
ITI	987.2	5.0	779.4	79.5	1,363.4	5.4	1,074.2	67.8

	MIXED NOISE							
	20%				35%			
	TIME	RANK	SIZE	ACC.	TIME	RANK	SIZE	ACC.
Rank	8.4	2.0	97.0	83.0	11.2	**2.0**	109.5	72.2
C4.5	**3.1**	**1.6**	**29.8**	82.4	**3.0**	2.1	33.0	73.0
NewID	7.3	3.0	190.9	81.5	6.3	3.0	340.3	69.6
CN2	42.9	–	96.0	**83.4**	20.8	–	**23.4**	**73.2**
IFM	15.9	2.0	141.4	83.1	18.8	**2.0**	212.4	69.8
ITI	1,337.6	3.2	426.7	70.9	1,436.6	3.7	545.6	58.5