The Design and Evaluation of a Rule Induction Algorithm

Kevin S. Van Horn and Tony R. Martinez
Computer Science Department
Brigham Young University
Provo, UT 84602

email: vanhorn@bert.cs.byu.edu, martinez@cs.byu.edu

technical report BYU-CS-93-11
June 1993

Keywords: machine learning, rule induction, noise, optimization, greedy algorithm, branch and bound.
Abstract: We present an algorithm for inductive learning from examples that outputs an ordered list of if-then rules as its hypothesis. The algorithm uses a combination of greedy and branch-and-bound techniques, and naturally handles noisy or stochastic learning situations. We also present the results of an empirical study comparing our algorithm with Quinlan's C4.5 on 1050 synthetic data sets. We find that BBG greatly outperforms C4.5 on rule-oriented problems, and equals or exceeds the performance of C4.5 on tree-oriented problems.

1 Introduction

Roughly speaking, the problem of inductive learning from examples is the following: Given a collection of *training examples* $(\bar{x}, c)$, where $\bar{x} \in X$ is a description of some object belonging to class $c \in C$, construct a *hypothesis* $h : X \to C$ for classifying arbitrary elements of X. The examples are presumed to be drawn independently and randomly from some distribution $\mathcal{D}$ over $X \times C$, and the goal is to minimize the misclassification error when the constructed hypothesis is tested on new examples drawn from the same distribution $\mathcal{D}$. Note that, as is a common assumption in the pattern-recognition literature [8], we assume that the relationship between a description $\bar{x}$ and its class c is in general a stochastic one, and not necessarily deterministic.

BBG is a learning algorithm we have developed for the case when $X = \{0,1\}^n$ for some n, i.e. there are n binary attributes. (In Section 9 we discuss how to handle arbitrary nominal (unordered) attributes.) BBG represents a hypothesis by a *rule list*, i.e. an ordered list of if-then rules which are tested in order until one is found that applies. The algorithm uses a combination of greedy techniques (successively insert the best new rule into the existing rule list) with branch-and-bound techniques (to find the best new rule); hence the acronym "BBG". It also uses a heuristic estimate of the actual error of the best rule list found within a given size bound, based on the size and empirical error, to trade off the conflicting goals of low empirical error and low hypothesis complexity. Experimental results assessing the efficacy of this heuristic are given.

In Section 7 we describe an approach to generating synthetic data sets in order to evaluate the average-case performance of learning algorithms. In Section 8 we give the results of testing BBG on 1050 separate synthetic data sets, comparing its performance with that of Quinlan's C4.5 tree induction algorithm. We find that on rule-oriented problems the average performance of BBG is much superior to that of C4.5, and on tree-oriented problems the average performance of BBG equals or exceeds that of C4.5.

2 Definitions

An *example* is a pair $(\bar{x}, c)$, where $\bar{x} \in X = \{0,1\}^n$ and $c \in C$ for some fixed n and finite set C; its *attribute vector* is $\bar{x}$ and its *class* is c. A learning algorithm is given a collection of examples (called the *training sample*) drawn independently and randomly from some unknown probability distribution $\mathcal{D}$ over $X \times C$. Note that it is possible for a training sample to contain multiple copies of the same example.

A training sample is *inconsistent* if it contains examples $(\bar{x}, c)$ and $(\bar{x}, c')$ with $c \neq c'$. It is *consistent* if it contains no such pair of examples. Inconsistent training samples can arise because $\mathcal{D}$ in general defines a stochastic relation between attribute vectors and classes.

A *rule* is a pair (T, c), where $c \in C$ and T is a conjunction of zero or more literals x_i or $\neg x_i$, $1 \leq i \leq n$. T is called the rule's *precondition*. The *size* of a rule (T, c), written $\mathrm{siz}(T, c)$, is one plus the number of conjuncts in T; thus the rule $(\neg x_1 \wedge x_5,\ c)$ has size 3.

The precondition T of a rule can be written as a vector $\bar{t} \in \{-, 0, 1\}^n$, where $t_i = 1$ if T contains the literal x_i, $t_i = 0$ if T contains the literal $\neg x_i$, and $t_i = -$ (don't care) otherwise. T may also be thought of as a predicate on binary n-vectors, and we write $T(\bar{v})$ for the predicate T applied to the vector $\bar{v}$. We say that an example $(\bar{v}, c)$ *matches* T, or rule (T, c'), if $T(\bar{v})$ is true (note that class is irrelevant).

A *hypothesis* is a function $h : \{0, 1\}^n \to C$. We say that h *misclassifies* an example $(\bar{x}, c)$ if $h(\bar{x}) \neq c$.

A *rule list* is a list of rules $(T_1, c_1) \cdots (T_k, c_k)$, with $T_k =$ **true** (the empty conjunction). This list represents the hypothesis h such that $h(\bar{x}) = c_i$, where $T_i(\bar{x})$ is true and $T_j(\bar{x})$ is false for all $j < i$. The *size* of a rule list is the sum of the sizes of its rules, i.e. it is the number of rules plus the total number of literals appearing in rules.

The *error* of a hypothesis h is the probability that $h(\bar{x}) \neq c$ when $(\bar{x}, c)$ is drawn at random from the same distribution $\mathcal{D}$ from which the training sample was drawn. The *empirical error* of a hypothesis is $\mathrm{err}(h, S)/|S|$, where S is the training sample and $\mathrm{err}(h, S)$ is the number of training examples misclassified by h.

3 Outline of the algorithm

BBG consists of two parts:

1. An algorithm G which, given a training sample, produces a sequence of rule lists of increasing size and decreasing empirical error. One can think of G as trying to minimize the empirical error for each of a series of size bounds, or equivalently, trying to minimize the size for each of a series of bounds on empirical error.

2. A method for choosing one of the rule lists output by G. There is a well-known trade-off between low empirical error and simplicity of hypothesis, as there is a tendency to fit noise or statistical flukes of the training sample when excessively complex hypotheses are allowed [1, 5, 9, 11].

A number of other machine-learning algorithms have used this two-part strategy; see, e.g., [11] or the literature on tree induction algorithms [2, 6].

The two parts are quite independent, and one can be changed without affecting the other. The second part is discussed in Section 6; for now we concentrate on algorithm G. In this section and Section 4 we assume that the training sample is consistent. We discuss how to deal with inconsistent training samples in Section 5.

Algorithm G is given in Figure 1. At its highest level of description G is a simple greedy algorithm. It successively inserts new rules into the growing rule list, at each step choosing the rule and insertion position which maximize the ratio (decrease in empirical error) / (rule size).

— Input: training sample S
— Output: sequence of hypotheses h_0, h_1, ...
$i := 0$
$h_0 := h :=$ the single rule $(\mathbf{true}, c)$, where c is the most common class in S
while $(\mathrm{err}(h, S) > 0)$
    $i := i + 1$
    $(T, c, p) := \mathrm{bestRule}(h, S)$
    insert rule (T, c) into h at position p
    $h_i := h$

Figure 1: Algorithm G

For a rule list of r rules there are r positions at which a new rule could be inserted, position p being immediately before rule p. (There is no point in putting a new rule after the final, default rule.) The procedure $\mathrm{bestRule}(h, S)$ tries to find a rule (T, c) and position p maximizing

$$(\mathrm{err}(h, S) - \mathrm{err}(h', S))/\mathrm{siz}(T, c),$$

where h' is obtained from h by inserting (T, c) at position p. The above is called the *gain-cost ratio* of (T, c, p).

4 Finding the best rule to insert

The difficult part, of course, is implementing bestRule(). We use a branch-and-bound algorithm with memory and time limits, which cannot guarantee optimality, but usually produces optimal or near-optimal results.

Branch-and-bound algorithms explore a search tree, the nodes of which correspond to sets of possible solutions, and the leaves of which are single solutions. The children of a node comprise a partition of the set of solutions represented by the node. The goal is to find a solution y maximizing $f(y)$, for some given function f. When a node z is reached one computes an upper bound u on the value of f for leaf nodes descended from z; if u is no greater than $f(y^\star)$ for the best solution $y^\star$ found so far, there is no need to explore the subtree rooted at z.

4.1 Preliminaries

For our purposes a node is a tuple $(\bar{t}, c, p)$, where $\bar{t} \in \{*, -, 0, 1\}^n$, c is a class, and p is a position to insert a rule. If $\bar{t}$ has no $*$ (undetermined) entries then the node is a leaf; otherwise it has three children obtained by choosing one index i for which $t_i = *$ and setting t_i to –, 0, or 1. We say that the node is *expanded on variable* i.

If $z = (\bar{t}, c, p)$ is a leaf node then $(\bar{t}, c)$ is a rule. If z is not a leaf node but $y = (\bar{t}', c, p)$ *is* a leaf node, then y is a descendant of z in the search tree if and only if $\bar{t}'$ may be obtained

```
Z := ∅; N := 0; g* := 0
initialize information needed by eval(), using h and S
for (each class c and position p)
    z := (*̄, c, p)
    (u, j) := eval(z)
    insert(u, j, z, Z)
while (Z ≠ ∅ and N < τ)
    (u, j, z) := best element of Z
    Z := Z − {(u, j, z)}
    while (u > g*)
        let (t̄, c, p) = z
        (z0, z1, z2) := children of z obtained by setting t_j to −, 1, and 0
        for (i := 0 to 2) (u_i, j_i) := eval(z_i)
        if (u1 > u2) swap (u1, j1, z1) and (u2, j2, z2)
        for (i := 0 to 1) insert(u_i, j_i, z_i, Z)
        (u, j, z) := (u2, j2, z2)
return y*
```

Figure 2: The procedure bestRule(h, S)

from $\bar{t}$ by replacing each $*$ entry with $-$, 0, or 1; this holds regardless of the method for choosing the variable on which to expand a node.

We write solns(z) for the set of all leaf nodes descended from z. Thus a node z corresponds to the set of possible solutions solns(z).

We define $z^\circ = (T, c, p)$, where T is obtained from $\bar{t}$ by replacing each $*$ entry with $-$. We write solns$^\circ(z)$ for solns$(z) - \{z^\circ\}$. As will be seen, considering z° separately from the other elements of solns(z) allows us to obtain a tighter upper bound on the gain-cost ratio.

Associated with bestRule() are constants μ (a memory bound) and τ (a time bound). It also uses variables Z (the collection of unexpanded nodes, with some associated information), N (the number of nodes evaluated so far), g^* (the gain-cost ratio of the best solution found so far), and y^* (the best solution found so far).

4.2 The branch-and-bound algorithm

The branch-and-bound algorithm for bestRule() is given in Figure 2, where we write $\bar{*}$ for the vector consisting of all $*$'s. The elements of Z are triples (u, j, z), where z is a node, u is an upper bound on the gain-cost ratio for elements of solns$^\circ(z)$, and j is the variable on which z is to be expanded.

The procedure eval(z), where $z = (\bar{t}, c, p)$, does the following:

1. It increments N, the number of nodes evaluated.

2. It computes an upper bound u on the gain-cost ratio for elements of solns$^\circ(z)$. (This is described in Section 4.3.) If z is a leaf node then solns$^\circ(z)$ is empty, so it sets $u = 0$.

3. As a side-effect of the upper-bound computation it produces two more values at little additional cost in computation: the variable j on which the node is to be expanded, and the gain-cost ratio g of the solution z°. If $g > g^\star$, then it sets $g^\star$ to g and $y^\star$ to z°.

4. The procedure returns (u, j) as its result.

Details on eval() are given in Section 4.3.

The procedure insert(u, j, z, Z) adds (u, j, z) to the collection Z if $u > g^\star$ and either

1. $|Z| < \mu$ (the memory bound has not been reached), or

2. (u, j, z) is better than the worst element w of Z (in which case w is removed to make room for (u, j, z)).

We consider (u, j, z) to be better than (u', j', z') if $u > u'$ (it has a better upper bound) or $u = u'$ and z contains fewer $*$'s than z' (z is closer to a leaf node than z'). We use this same definition of "better" when we choose the best element of Z in the first element of the while loop in Figure 2.

Note that bestRule() uses a combination of best-first search (to concentrate on the most promising nodes) and depth-first search (to quickly reach some leaf). The most promising node is removed from Z, and then a linear path downward from that node is explored, always adding literals to the precondition, until we have determined that further exploration along the path is fruitless ($u \leq g^\star$). No leaf is ever expanded because the upper bound $u = 0$ associated with it is never more than $g^\star$. As each node z is reached, the solution z° is considered and evaluated. Once it is decided to stop exploring a path, the (new) most promising node is removed from Z and the process repeats.

4.3 Computing the upper bound

We now give the details of eval(). Let us define the following:

- $\Delta(y) = \text{err}(h, S) - \text{err}(h', S)$, where $y = (T, c, p)$ is a leaf node, S is the collection of training examples, h is the current hypothesis, and h' is obtained from h by inserting the rule (T, c) at position p.

- $\text{msiz}(z) = \text{siz}(T, c)$, where z is a node and $z^\circ = (T, c, p)$, i.e. $\text{msiz}(z)$ is the minimum size that any node derived from z will have.

The gain-cost ratio of a leaf y is just $\Delta(y)/\text{siz}(y)$. Thus if δ is an upper bound on $\Delta(y)$ for $y \in \text{solns}^\circ(z)$ then $\delta/\text{msiz}(z)$ is an upper bound on the gain-cost ratio for elements of $\text{solns}^\circ(z)$.

Let us consider the computation of $\Delta(T, c, p)$ for a rule (T, c) and position p. We define the following:

- S_p = those examples in S that match none of the first $p - 1$ rules of h.

- S_p^T = those examples in S_p that match T.

- $G(T, c, p)$ = those examples $(\bar{x}, c')$ in S_p^T for which $c' = c$ and $h(\bar{x}) \neq c'$ (the "good" examples).
- $G(z) = G(z^\circ)$, where z is a non-leaf node.
- $B(T, c, p)$ = those examples $(\bar{x}, c')$ in S_p^T for which $c' \neq c$ and $h(\bar{x}) = c'$ (the "bad" examples).

$G(T, c, p)$ is the collection of previously *misclassified* examples which are *correctly classified* after inserting rule (T, c) at position p. $B(T, c, p)$ is the collection of previously *correctly-classified examples* which are *misclassified* after inserting the new rule. Then $\Delta(T, c, p) = |G(T, c, p)| - |B(T, c, p)|$, i.e. the number of old errors corrected minus the number of new errors introduced.

Let $y = (T, c, p) \in \text{solns}^\circ(z)$ and $z^\circ = (T', c, p)$. Then $S_p^T \subseteq S_p^{T'}$ (since T contains every literal that T' does), hence $G(y) \subseteq G(z^\circ) = G(z)$. Then $|G(z)| \geq |G(y)| \geq \Delta(y)$ for every $y \in \text{solns}^\circ(z)$, and we obtain $|G(z)|/\text{msiz}(z)$ as an upper bound on the gain-cost ratio for elements of $\text{solns}^\circ(z)$.

We can improve this bound without increasing the asymptotic time complexity of eval(z), at the same time computing the gain-cost ratio of z° and identifying a promising variable on which to expand z. bestRule() precomputes the various S_p by ordering the examples in S according to the first rule in h which they match, and computing for each p the index $f[p]$ of the first example matching none of the first $p-1$ rules of h. If there are m examples then S_p is just $S[f[p]]$ through $S[m]$. bestRule() also precomputes an array K such that $K[i] = h(\bar{x})$, where $(\bar{x}, c')$ is the i-th example. Letting $z = (\bar{t}, c, p)$ and $z^\circ = (T, c, p)$, $|G(z)|$ is computed by stepping through the examples $S[f[p]]$ through $S[m]$, checking to see if each example $S[i] = (\bar{x}, c')$ matches T, and if so, comparing c' to $K[i]$ and c to determine if $S[i] \in G(z)$. This test of example i takes $\Theta(n)$ worst-case time, where n is the number of binary attributes. With $\Theta(1)$ additional computation time eval() determines if $S[i] \in B(z^\circ)$; summing over the examples this gives $|B(z^\circ)|$, which combined with $|G(z)| = |G(z^\circ)|$ and $\text{msiz}(z^\circ)$ gives the gain-cost ratio for z°.

Eval() also use counters $\kappa[i, b]$ $(1 \leq i \leq n,\ b \in \{0, 1\})$ initialized to 0. Once an example $(\bar{x}, c')$ has been identified as an element of $G(z)$, eval() uses $\Theta(n)$ time to step through the entries of $\bar{x}$, incrementing $\kappa[i, x_i]$ whenever $t_i = *$. This does not increase the asymptotic time complexity. In essence, for each i such that $t_i = *$, eval() splits the elements of $G(z)$ into those for which $x_i = 0$ and those for which $x_i = 1$, thus computing $|G(z_{ch})|$ for each possible child z_{ch} of z formed by adding another literal. Let $z[i : b]$ be the possible child of z obtained by replacing t_i with b. Then after all the examples in S_p have been processed we have $\kappa[i, b] = |G(z[i : b])|$ for all i such that $t_i = *$. Since $\text{msiz}(z[i : b]) = \text{msiz}(z) + 1$, we then have $\kappa[i, b]/(\text{msiz}(z) + 1)$ as an upper bound on the gain-cost ratio for elements of $\text{solns}(z[i : b])$. Taking the maximum over $b \in \{0, 1\}$ and all i for which $t_i = *$ gives an upper bound on the gain-cost ratio for elements of $\text{solns}^\circ(z)$. Furthermore, the i which attains this maximum is an obvious choice for the variable on which to expand z in order to quickly reach a good solution in the depth-first part of our search.

Putting all of this together gives us the algorithm shown in Figure 3.

```
let (t̄, c, p) = z and (T, c, p) = z°
γ := 0; β := 0
init all κ[i, b] to 0
for (j := f[p] to m) if (S[j] matches T)
    let (x̄, c′) = S[j]
    if (c′ ≠ c and c′ = K[j]) increment β          — S[j] ∈ B(z°)
    if (c′ = c and c′ ≠ K[j])                      — S[j] ∈ G(z°)
        increment γ
        for (i := 1 to n) if (t_i = ∗) increment κ[i, x_i]
g := (γ − β)/msiz(z°)                              — gain-cost ratio of z°
if (g > g*)
    g* := g; y* := z°
d := 0
for (i := 1 to n and b := 0 to 1) if (κ[i, b] > d)
    d := κ[i, b]; j := i
u := d/(msiz(z) + 1)
N := N + 1                                         — increment # of nodes evaluated
return (u, j)
```

Figure 3: The procedure eval(z)

4.4 Time complexity of bestRule(h, S)

Let m be the number of examples in S, r the number of rules in h, k the number of classes, and n the number of binary attributes.

Each call to eval() takes $\Theta(mn)$ worst-case time. There are at least kr calls to eval(), which take place when Z is being initialized. If $kr < \tau$ then there are at most $\tau + 3n - 1$ calls to eval()—at the top of the outer while loop we have $N < \tau$, hence $N \leq \tau - 1$, and there are at most $3n$ total calls to eval() in the inner loop. Thus we have $\Theta(kr + \tau + n)$ calls to eval() in the worst case.

Since Z is implemented using a pair of heap structures serving as priority queues, each call of insert() takes $\Theta(\log \mu)$ worst-case time, and to remove an element of Z takes $\Theta(\log \mu)$ worst-case time. There are kr calls to insert() when Z is being initialized. There are upwards of $\max(0, \tau + 2n - 1)$ total additional calls to insert(), figuring as in the preceding paragraph. Thus we have $\Theta(kr + \tau + n)$ calls to insert() in the worst case. Since every element removed from Z must have been inserted earlier, there are $O(kr + \tau + n)$ removals of elements from Z. Thus we have $\Theta(kr + \tau + n)$ calls to insert() or removals from Z in the worst case.

If we use a bucket sort, the sorting of S and initialization of $f[]$ to produce the various S_p takes $\Theta(mnr)$ time. Initializing $K[]$ then takes an addition $\Theta(m)$ time.

If the training sample is inconsistent, then sorting S (to identify the clans) and then reducing the clans takes $\Theta(nm \log m + km)$ time.

We will make the following assumptions:

1. $\mu = O(2^{mn})$ (and hence $\log \mu = O(mn)$). It certainly makes no sense to have the memory bound grow at a rate that is exponential or greater in mn, so this is a safe assumption.

2. $m = O(2^n)$ (and hence $\log m = O(n)$). Otherwise the number of training examples grows large compared to the total number of possible attribute vectors.

Then the total worst-case time of bestRule() is

$$\begin{aligned}
&\Theta((kr + \tau + n)mn) + \Theta((kr + \tau + n)\log \mu) + \Theta(mnr) + \Theta(nmlogm + km) \\
&= \Theta((kr + \tau + n)mn) + O((kr + \tau + n)mn) + \Theta(mnr) + O(nmn) + \Theta(km) \\
&= \Theta((kr + \tau + n)mn).
\end{aligned}$$

4.5 Termination of G

For algorithm G to terminate we need to guarantee that bestRule(h, S) returns a solution with positive gain-cost ratio whenever $\mathrm{err}(h, S) > 0$, so that $\mathrm{err}(h, S)$ decreases. We can guarantee this as long as $\tau > |C| + 3n$ (time bound sufficiently large to allow a leaf node to be reached.)

To see this, first note that whenever $\mathrm{err}(h, S) > 0$ there is some leaf node with positive gain-cost ratio, viz. $(\bar{x}, c, 1)$ where $(\bar{x}, c)$ is any example misclassified by h. This solution has a gain-cost ratio of $o/n > 0$, where o is the number of occurrences of $(\bar{x}, c)$ in S. Thus at least one of the nodes initially created—in fact, the first chosen for expansion—will have a positive upper bound. The depth-first search starting from this node will continue (at least) until either

1. $g^\star > 0$ (we have found a solution with positive gain-cost ratio), or

2. $g^\star = 0$ and we reach a node whose upper bound is 0.

We will show that (2) cannot happen.

The initial depth-first search starts from a node with no '–' entries ($(\bar{*}, c, p)$ for some c and p), and at each step the search continues with a child obtained by replacing a $*$ with 0 or 1. Thus no node expanded in the initial depth-first search contains a '–' entry. We show that (2) above cannot happen by showing that whenever a positive upper bound is computed for a node z with no '–' entries, either

a. we will compute a positive upper bound for the child z_{ch} of z next expanded, or

b. some child of z is a leaf with positive gain-cost ratio.

If (b) occurs then it will cause $g^\star > 0$, since eval() is called for each child of z when z is expanded.

It is clear from inspection of eval() that if z is a non-leaf node, a positive upper bound is computed for z if and only if $G(z)$ is nonempty. Now suppose an upper bound $u > 0$ and variable j on which to expand are computed for a node z with no '–' entries. Then z is not a leaf node, and $u = |G(z[j : b])| > 0$ for some $b \in \{0, 1\}$, i.e. $G(z[j : b])$ is nonempty. There are two cases:

- $z[j:b]$ is not a leaf node. Then a positive upper bound is computed for $z[j:b]$ (since $G(z[j:b])$ is nonempty). Since z_{ch} is either $z[j:0]$ or $z[j:1]$—whichever has a higher upper bound—this tells us that a positive upper bound is computed for z_{ch}, and (a) above holds.

- $z[j:b]$ is a leaf node. Since $G(z[j:b])$ is nonempty, it contains at least one example $(\bar{x}, c) \in S_p$. This example is misclassified by the current rule list h and matches $\bar{t}$, where $z[j:b] = (\bar{t}, c, p)$. In fact, $\bar{x} = \bar{t}$, since $\bar{t}$ contains no '–' entries. Then $\Delta(z[j:b]) = \Delta(\bar{x}, c, p) = o$, where o is the number of occurrences of $(\bar{x}, c)$ in S, hence $z[j:b]$ has a gain-cost ratio of $o/n > 0$, and (b) above holds.

5 Inconsistent training samples

In sections 3 and 4 we assumed that the training sample was consistent. Two simple modifications suffice for dealing with inconsistent training samples. The first and most obvious modification is that the test "$\mathrm{err}(h, S) > 0$" in the top level of algorithm G should be replaced by "$\mathrm{err}(h, S) > E$", where E is the minimum number of misclassified training examples that can be achieved by any hypothesis. E can be computed as follows. Partition the training sample into maximal groups of examples with the same attribute vector; such a group we call a *clan*. Let E_i^c be the number of examples in clan i whose class is *not* c, and compute $E_i = \min\{E_i^c : c \in C\}$. Then $E = \sum_i E_i$.

The second modification involves bestRule(). The termination proof given in Section 4.5 breaks down for inconsistent samples, because it is possible to have a misclassified example $(\bar{x}, c) \in S_p$ and yet have $\Delta(\bar{x}, c, p) \leq 0$. This is because the gain from correcting the misclassification of $(\bar{x}, c)$ may be offset by a loss for misclassifying examples $(\bar{x}, c')$, $c' \neq c$, that were previously correctly classified.

We can fix this problem with a preprocessing step. For each class c, we define $\mathrm{reduce}(S, c)$ to be the collection of examples obtained from S by reducing each clan as follows. Let $\bar{x}$ be the attribute vector defining the clan, and let $c' = h(\bar{x})$. If $c' = c$, remove the entire clan. Otherwise, remove from the clan all examples whose output class is neither c nor c'. Then repeatedly remove one example $(\bar{x}, c)$ and one example $(\bar{x}, c')$ until all examples in the clan have the same class.

$\mathrm{reduce}(S, c)$ is a consistent collection of examples, since all members of a clan have the same class. Furthermore, for any possible solution $y = (T, c, p)$ the gain-cost ratio of y is the same whether computed using S or using $\mathrm{reduce}(S, c)$. Proof: Consider a clan with attribute vector $\bar{x}$ and $h(\bar{x}) = c'$. If $c' = c$ then $\Delta(y)$ does not depend on the examples in the clan (their classification is unchanged by y). Otherwise, $\Delta(y)$ depends only on those examples in the clan with class c or c', as the remainder are misclassified by h and not corrected by y. In addition, each example $(\bar{x}, c)$ adds one to $\Delta(y)$ (it is an element of $G(y)$), and each example $(\bar{x}, c')$ subtracts one from $\Delta(y)$ (it is an element of $B(y)$); thus if we remove one of each $\Delta(y)$ is unaffected.

So we precompute $\mathrm{reduce}(S, c)$ for each $c \in C$ at the beginning of bestRule(), then use $\mathrm{reduce}(S, c)$ in place of S when evaluating a node $z = (\bar{t}, c, p)$.

6 Trading off empirical error and hypothesis complexity

Algorithm G produces a sequence of hypotheses $h_0, h_1, \ldots$ of increasing complexity and decreasing empirical error. In this section we describe how BBG trades off the conflicting goals of low empirical error and low complexity to choose one of the h_i output by G. In the literature can be found a number of methods for handling this trade-off; these include cross-validation [2, 11], using a separate hold-out set on which to test the sequence of hypotheses produced [3], the minimum description-length principle [7], Vapnik's structural risk minimization [9, 10], and heuristic estimates of the actual error of the best hypothesis found for a given size bound [6]. Any of these methods can be used in combination with algorithm G.

BBG uses the last approach mentioned. The various h_i are compared according to a heuristic estimate of their actual error, computed from their empirical error and size, and the best is chosen. (This method bears some resemblance to Vapnik's structural risk minimization [9, 10].) In particular let s_i be the size of h_i and $e_i = \mathrm{errs}(h_i, S)$. BBG chooses the rule list h_i for which $\mathrm{estErr}(s_i, e_i)$ is smallest, where $\mathrm{estErr}(s, e)$ is our heuristic estimate of the actual error of the best rule list found of size at most s, given that it misclassifies e of the training examples.

We now describe $\mathrm{estErr}(s, e)$. In what follows let m be the number of training examples, n the number of attributes, and R the set of hypotheses defined by rule lists of size at most s.

Consider the probability $\pi(m, e, \epsilon, |R|)$ that at least one hypothesis in R misclassifies ϵ or fewer examples, under the following simplifying (but unrealistic) assumptions:

1. All hypotheses in R have the same error ϵ.
2. The number of examples misclassified by one hypothesis is completely independent of the number misclassified by any other hypothesis.

We then define

$$\mathrm{estErr}(s, k) = (\text{that } \epsilon \text{ for which } \pi(m, e, \epsilon, \tilde{N}) = \alpha)$$

(typically we set $\alpha = 0.5$), where $\tilde{N}$ is an estimate of $|R|$. Since $\pi()$ is monotonically decreasing in ϵ, we can compute $\mathrm{estErr}(s, e)$ using a binary search of the interval $[0, 1]$. It remains only to discuss the computation of $\pi()$ and $\tilde{N}$.

For any single hypothesis h' with error ϵ, the number of training examples misclassified out of m total examples is a random variable following the binomial distribution $\mathrm{Binom}(m, \epsilon)$. Thus there is a probability of

$$\mathrm{LB}(m, e, \epsilon) = \sum_{i=0}^{e} \binom{m}{i} \epsilon^i (1 - \epsilon)^{m-i}$$

that h' will misclassify at most e out of m randomly and independently drawn training examples.

Given N different hypotheses, let ξ_i $(1 \leq i \leq N)$ be the number of training examples misclassified by hypothesis i. Under the assumptions 1 and 2 above, the ξ_i are N independent random variables, each distributed as Binom(m, ϵ). So

$$\begin{aligned}
\pi(m, e, \epsilon, N) &= \mathbf{Pr}[\bigvee_{i=1}^{N} (\xi_i \leq e)] \\
&= 1 - \mathbf{Pr}[\bigwedge_{i=1}^{N} \neg(\xi_i \leq e)] \\
&= 1 - (1 - \mathbf{Pr}[\xi_1 \leq e])^N \\
&= 1 - (1 - \mathrm{LB}(m, e, \epsilon))^N
\end{aligned}$$

The estimate $\tilde{N}$ of $|R|$ is computed as follows. A nontrivial rule list of size s can have at most $b(s) = \lfloor (s-1)/2 \rfloor$ non-default rules—if it has more than this then some rule other than the final, default rule must have size 1, meaning that its precondition is the empty conjunction (**true**), and no rule following it is ever reached. A rule list with r non-default rules can have no more than a total of nr literals, and so has a size of at most $r + 1 + nr$. Applying some algebra, this tells us that a rule list of size s has at least $a(s) = \lceil (s-1)/(n+1) \rceil$ rules.

If there are only two classes, then there are a total of $2^s \binom{nr}{l}$ different rule lists of size s with r rules and $l = s - r - 1$ literals. This gives us a total of

$$N_0 = \sum_{t=1}^{s} 2^t \sum_{r=a(t)}^{b(t)} \binom{nr}{t-r-1}$$

rule lists of size at most s. $|R|$ will actually be smaller than this, because several different rule lists may represent the same hypothesis. For example, a rule list may have a non-default rule with the precondition **true** (even given the requirement $r \leq b(t)$), making it equivalent to a variety of smaller rule lists. With this in mind we set

$$\tilde{N} = N_0^{\beta}$$

for some $0 < \beta < 1$.

The above estimate of $|R|$ assumed there were only two classes. Theoretical results using the Vapnik-Chervonenkis dimension to bound the difference between empirical error and actual error [1], applied to conjunctive concepts [4], suggest that this difference does not depend on the number of classes. Thus we compute $\tilde{N}$ as described above even when there are more than two classes.

The reader may be concerned at the unrealistic assumptions and rough approximations used in the derivation of estErr(). In fact, our experience has been that estErr() tends to greatly overestimate the actual errors of the rule lists G produces. What matters, though, is how well estErr() works at picking the best rule list from the sequence G produces. We give evidence in Section 8 that it works rather well at this task.

7 Evaluation strategy

We have evaluated the performance of BBG by testing it on a large number of synthetic data sets. The data sets were produced using two problem generators we developed: rgenex (for

rule-oriented problems) and tgenex (for tree-oriented problems). These generators take the parameters n (number of attributes), k (number of classes), η (noise level), m (size of training sample), m' (size of test sample), and others described below. We require $0 \leq \eta < 0.5$. In outline, the problem generators work as follows:

1. A *target* h is randomly generated. For rgenex h is a rule list, and for tgenex h is a decision tree; for both we have n Boolean attributes and k classes $C = \{1, \ldots, k\}$.

2. A probability distribution $\mathcal{D}_X$ over $X = \{0,1\}^n$ is constructed.

3. A distribution $\mathcal{D}$ over $X \times C$ is obtained from $\mathcal{D}_X$ and h by adding a level η of uniform noise. Specifically, the probability of $(\bar{x}, c)$ under $\mathcal{D}$ is pq, where p is the probability of $\bar{x}$ under $\mathcal{D}_X$, $q = 1 - \eta$ if $c = h(\bar{x})$, and $q = \eta/(k-1)$ if $c \neq h(\bar{x})$.

4. The m training examples and m' testing examples are randomly and independently selected from the distribution $\mathcal{D}$.

Note that the target h has an error of η on the distribution $\mathcal{D}$, and that this is the minimum error achievable by any hypothesis.

For any fixed setting of the parameters to rgenex and tgenex one can investigate the expected error of a learning algorithm as follows: generate a series of data sets using these parameters, run the algorithm on the training samples and average the resultant error rates on the test samples.

7.1 Rgenex

Rgenex takes additional parameters r and l, which specify the number of non-default rules and total number of literals the target h must have. The target h is generated as follows:

1. Classes are (nearly) evenly distributed among the $r+1$ rules. Each class is used between $\lfloor (r+1)/k \rfloor$ and $\lceil (r+1)/k \rceil$ times, with the $(r+1)$ mod k classes that occur an extra time being chosen randomly without replacement from a uniform distribution over C. The assignment of classes to rules is obtained by initializing an array of length $r+1$ with the appropriate numbers of each class, then randomly permuting the array, with all permutations being equally likely.

2. The literals are randomly chosen by selecting l pairs (i,j) randomly without replacement from a uniform distribution over $\{1, \ldots, r+1\} \times \{1, \ldots, n\}$. For each pair (i,j) we add either x_j or $\neg x_j$ to the precondition of rule i, the choice being made at random with equal probabilities.

3. Rule lists with superfluous rules are disallowed. A rule is superfluous whenever its precondition is true the precondition of some preceding rule is also true.

Let T_i be the precondition of rule i of the target h, and let W_i be the set of $\bar{x} \in X$ such that $T_i(\bar{x})$ is true and $T_j(\bar{x})$ is false for $j < i$. The distribution $\mathcal{D}_X$ is defined by the following two properties:

if $(s = 1)$ return leaf
if $(s = 2)$ return mktree(rand(A), leaf, leaf)
(u, l) := setBounds$(s, |A|)$
ls := rand$(\{l, \ldots, u\})$; $rs := s - ls$
i := rand(A); $A' := A - \{i\}$
return mktree$(i,$ rtree$(ls, A'),$ rtree$(rs, A'))$

Figure 4: The procedure rtree(s, A)

1. The rules are equally likely to be chosen, i.e. if $\bar{x}$ is randomly selected according to $\mathcal{D}_X$ then $\mathbf{Pr}[\bar{x} \in W_i] = 1/(r+1)$ for all i.

2. The distribution is uniform "within" each rule, i.e. for each i the elements of W_i are equally likely.

7.2 Tgenex

Tgenex takes the additional parameter s, which specifies the size (number of leaves) of the target h. The structure of the target h (h with class labels omitted from the leaves) is generated by calling rtree$(s, \{1, \ldots, n\})$, where rtree() is defined in Figure 4. Mktree(i, t_1, t_2) returns the decision tree whose root node is labeled x_i, with left subtree t_1 and right subtree t_2. Rand(F), for any finite set F, returns an element of F randomly selected from a uniform distribution. We write leaf for an unlabeled leaf. The call to setBounds() guarantees that both ls and rs are between 1 and $2^{|A|-1}$ inclusive. This ensures that each subtree has at least one node, and that we won't run out of attributes ($A = \emptyset$) on a recursive call of rtree().

Leaf nodes are labeled as follows. If the leaf node is not the right sibling of a pair of sibling leaf nodes then its class is randomly chosen from a uniform distribution over C. Otherwise the leaf node's class is randomly chosen from a uniform distribution over $C - \{c\}$, where c is the class of its sibling leaf node. (There is no point in having two sibling leaf nodes with the same class, as the tree could be simplified by deleting them and labeling their parent with the common class.) In addition, labelings in which some class labels more than twice as many leaves as another class are disallowed.

Let W_i be the set of $\bar{x} \in X$ satisfying each of the tests on the path from the root of h to leaf i. The distribution $\mathcal{D}_X$ is defined by the following two properties:

1. The leaves are equally likely to be chosen, i.e. if $\bar{x}$ is randomly selected according to $\mathcal{D}_X$ then $\mathbf{Pr}[\bar{x} \in W_i] = 1/s$ for all i.

2. The distribution is uniform "within" each leaf, i.e. for each i the elements of W_i are equally likely.

l	r	n	k	η	BBG	c4.5	c4.5rules
29	12	38	3	0.03	1.85 ± 0.84	9.47 ± 1.79	8.31 ± 1.49
32	9	38	5	0.02	1.38 ± 0.51	12.20 ± 2.04	9.76 ± 1.57
30	11	45	5	0	0.45 ± 0.26	11.41 ± 1.58	8.30 ± 1.62
34	7	50	4	0.04	3.56 ± 0.77	13.22 ± 2.42	13.21 ± 2.20
34	7	52	2	0.05	4.05 ± 1.01	8.17 ± 1.42	7.44 ± 1.32
30	11	54	6	0	0.58 ± 0.30	12.57 ± 1.82	9.31 ± 1.58
32	9	61	4	0.02	2.19 ± 0.82	12.50 ± 1.78	10.99 ± 1.73
36	5	64	3	0.05	6.21 ± 0.93	6.75 ± 1.43	8.17 ± 1.41
31	10	67	3	0.04	2.97 ± 0.80	13.73 ± 2.11	12.10 ± 2.14
28	13	79	2	0.02	4.36 ± 1.02	6.39 ± 1.50	5.30 ± 1.42

Table 1: Comparison of BBG and C4.5 on rule-oriented problems

8 Experimental results

We tested BBG on 1050 synthetic data sets generated by rgenex and tgenex. We wanted to compare its performance on the same data sets with that of some well-respected and widely-known learning algorithm; for this purpose we chose Quinlan's C4.5. We compiled the code that came with his book [6] on DECstation 5000's and HP 710's. Since C4.5 can produce either decision trees (the c4.5 program) or rule lists (the c4.5rules program), we tested both alternatives. The default parameters were always used with these programs.

For BBG we set the memory limit μ and time limit τ as follows. Let m be the number of examples and n the number of Boolean attributes. For the 500 data sets summarized in Table 1 we set μ to $2mn$. For the 550 data sets summarized in Table 2 we set μ to $\max(50000, 2mn)$. In both cases we set τ to $\max(250000, 10mn)$. (For $n = 50$ and $m = 500$ we get $2mn = 50000$ and $10mn = 250000$.) For estErr() we set $\alpha = \beta = 0.5$. Typical execution times on an HP 710 workstation with 32 MB of memory and little else running on the machine were around 25 minutes for $m = 500$, but varied significantly from one data set to another.

We generated 50 data sets for each of 10 different parameter settings of rgenex (a total of 500 data sets), and compared BBG with C4.5 on these. We always had $m = 500$, $m' = 30000$, and $r + l = 41$, giving a size of 42 for all of our target rule lists. Beyond this, we came up with the 10 parameter settings by choosing them at random.

The results are summarized in Table 1. The column labeled BBG gives the amount by which the average error of BBG exceeds η, and the columns labeled c4.5 and c4.5rules give the amounts by which the average errors of the corresponding algorithms exceed that of BBG. Each of these amounts is in percent, with a 95% confidence interval computed. For each of the ten settings of parameter values for rgenex, BBG outperforms both c4.5 and c4.5rules by wide margins.

We next generated 50 data sets for each of 11 different parameter settings of tgenex (a total of 550 data sets), and compared BBG with C4.5 on these. We set $s = 20$, $m = 500$, and $m' = 30000$ in all cases. The first ten parameter settings used the same values for n, k,

s	n	k	η	BBG	c4.5	c4.5rules
20	38	3	0.03	2.33 ± 0.56	8.94 ± 1.94	3.25 ± 1.22
20	38	5	0.02	0.69 ± 0.28	2.87 ± 1.14	0.86 ± 0.67
20	45	5	0	0.58 ± 0.32	3.88 ± 1.30	0.09 ± 0.35
20	50	4	0.04	1.43 ± 0.49	6.33 ± 1.72	3.13 ± 1.16
20	52	2	0.05	10.69 ± 1.90	14.46 ± 2.49	12.84 ± 2.63
20	54	6	0	0.76 ± 0.43	1.63 ± 0.84	-0.18 ± 0.48
20	61	4	0.02	1.77 ± 0.54	7.01 ± 2.02	2.14 ± 1.38
20	64	3	0.05	4.68 ± 0.96	12.93 ± 2.78	8.40 ± 2.58
20	67	3	0.04	4.66 ± 1.36	12.22 ± 2.82	6.24 ± 2.42
20	79	2	0.02	12.95 ± 2.63	15.79 ± 2.78	12.98 ± 3.20
20	50	2	0	3.33 ± 0.93	21.31 ± 2.38	14.71 ± 2.49

Table 2: Comparison of BBG and C4.5 on tree-oriented problems

r	l	n	k	η	m	BBG-O	DIFF
16	5	50	2	0.05	500	6.05	0.13 ± 0.14
32	9	30	2	0.05	500	8.44	0.44 ± 0.24
32	9	50	2	0	500	2.89	0.38 ± 0.25
32	9	50	2	0.05	250	18.22	3.48 ± 0.69
32	9	50	2	0.05	500	9.07	0.94 ± 0.51
32	9	50	6	0.05	500	6.23	0.37 ± 0.30
50	15	50	2	0.05	500	15.34	1.18 ± 0.60
32	9	50	2	0.05	750	7.22	0.30 ± 0.18
32	9	80	2	0.05	500	7.65	1.19 ± 0.65

Table 3: Comparison of BBG and BBG-O

and η as we used for rgenex. An eleventh parameter setting was added with $n = 50$, $k = 2$, and $\eta = 0$.

The results are summarized in Table 2 (the last three columns have the same meaning as before). In each case the average performance of BBG exceeds that of c4.5 and either is very close to or exceeds that of c4.5rules. BBG's performance advantage is greater when there are few classes, and is quite striking when $k = 2$ and $\eta = 0$ (the last case).

One additional experiment tested the performance of our heuristic for choosing one of the hypotheses output by G. Let BBG-O be a variant of BBG in which we replace this heuristic with an oracle that always picks the best hypothesis. (This is implemented by letting BBG-O peek at the test sample so that it can pick from the sequence of rule lists output by G, that which has the lowest error on the test sample.) We generated 30 data sets for each of 9 different parameter settings of rgenex (270 data sets total). Each parameter setting had $m' = 30000$. On each data set we ran both BBG and BBG-O. We set τ, μ, α, and β as for the experiment summarized in Table 2. The results are summarized in Table 3. The column

labeled BBG-O gives the average error of BBG-O (in percent), and the column labeled DIFF gives the amount by which the average error of BBG exceeds that of BBG-O (in percent), with a 95% confidence interval computed.[1] These results indicate that our heuristic works rather well, in spite of the questionable assumptions used to derive it.

9 Extending BBG to handle nominal attributes

We can extend BBG to handle arbitrary nominal attributes in either of two ways, depending on what kinds of rule preconditions we wish to allow. In either case the size of a rule is still one plus the number of conjuncts in its precondition. For simplicity we assume that each attribute i takes values from $X_i = \{0, \ldots, d_i - 1\}$ for some positive integer d_i.

A literal x_i is equivalent to either of the tests $x_i = 1$ or $x_i \neq 0$, and $\neg x_i$ is equivalent to either of $x_i = 0$ or $x_i \neq 1$. The first possibility then is to allow as a precondition any conjunction of tests of the form $x_i = v$ or $x_i \neq v$, where $v \in X_i$. In this case we can just transform each attribute i into a sequence of Boolean attributes $j_0, \ldots, j_{d_i - 1}$, with attribute j_v being true if and only if the original attribute i has value v, and apply BBG to the result. The output of BBG is transformed back by replacing each literal x_{j_v} with $x_i = v$ and $\neg x_{j_v}$ with $x_i \neq v$.

The second possibility is to allow as preconditions only conjunctions of tests of the form $x_i = v$ for $v \in X_i$. If we use the transformation of the preceding paragraph, we then need a modification of BBG in which only positive literals are allowed. We obtain this via the following modifications to bestRule():

1. A node is now a tuple $(\bar{t}, c, p)$, where $\bar{t} \in \{*, -, 1\}^n$. Each non-leaf node now has only two children, instead of three.

2. The body of the inner while loop ("while $(u > g^\star)$") becomes

   let $(\bar{t}, c, p) = z$
   (z_0, z_1) := children of z obtained by setting t_j to – and 1
   for (i := 0 to 1) (u_i, j_i) := eval(z_i)
   insert(u_0, j_0, z_0, H)
   $(u, j, z) := (u_1, j_1, z_1)$

3. In the upper-bound computation, counters $\kappa[i, 0]$ and $\kappa[i, 1]$ are replaced by a single counter $\kappa[i]$, for each Boolean attribute i. The statement "increase $\kappa[i, x_i]$" is replaced by "if (x_i) increase $\kappa[i]$". All references to $\kappa[i, b]$ are replaced by $\kappa[i]$.

10 Acknowledgements

This research was supported in part by grants from Novell and Wordperfect.

[1] The confidence intervals for the first, third and sixth rows of the table are questionable, as the computed averages are less than three standard errors from 0, the minimum possible value for DIFF.

References

[1] Blumer, A., et al. (1989.) Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM* 36, 929–965.

[2] Breiman, L., et al. (1984.) *Classification and Regression Trees.* Belmont, CA: Wadsworth.

[3] Devroye, L. (1988.) Automatic pattern recognition: a study of the probability of error. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10, 530–543.

[4] Haussler, D. (1988.) Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence* 36, 177–221.

[5] Haussler, D. (1992.) Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation* 100, 78–150.

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

[7] Quinlan, J. R., & Rivest, R. L. (1989.) Inferring decision trees using the Minimum Description Length Principle. *Information and Computation* 80, 227–248.

[8] Tou, J., & Gonzalez, R. (1974.) *Pattern Recognition Principles.* Reading, MA: Addison-Wesley.

[9] Vapnik, V. N. (1982.) *Estimation of Dependences Based on Empirical Data.* New York: Springer-Verlag.

[10] Vapnik, V. N. (1989.) Inductive principles of the search for empirical dependences. In *Proceedings of the 2nd Annual Workshop on Computational Learning Theory.* San Mateo, CA: Morgan Kaufmann.

[11] Weiss, S., & Kulikowski, C. (1991.) *Computer Systems That Learn.* San Mateo, CA: Morgan Kaufmann.