The BBG Rule Induction Algorithm

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

This paper appeared in *Proceedings of the 6th Australian Joint Conference on Artificial Intelligence*, Melbourne, Australia, 17 Nov. 1993, pp. 348–355.

The BBG Rule Induction Algorithm

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

Abstract: We present an algorithm (BBG) for inductive learning from examples that outputs a rule list. BBG 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 BBG 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 C4.5's performance on tree-oriented problems.

1 Background

BBG is an algorithm for learning to classify vectors of binary attributes. (It is straightforward to extend it to arbitrary nominal attributes.[7]) BBG outputs a rule list, i.e. an ordered list of if-then rules which are tried in order until one is found that applies. In this section we introduce the basic concepts and vocabulary relevant to BBG.

An *example* is a pair $(\mathbf{x}, c)$, where $\mathbf{x} \in X = \{0,1\}^n$ and $c \in C$ for some fixed n and finite set of *classes* C. A *hypothesis* is a function $h : X \to C$. We say that h *misclassifies* an example $(\mathbf{x}, c)$ if $h(\mathbf{x}) \neq c$.

A *rule* is a pair $(\mathbf{t}, c)$, where $c \in C$ and $\mathbf{t} \in \{-, 0, 1\}^n$; $\mathbf{t}$ is called the rule's *precondition.* We say that a vector $\mathbf{x}$ (or example $(\mathbf{x}, c)$) *matches* $\mathbf{t}$ (or rule $(\mathbf{t}, c')$) if $x_i = t_i$ whenever t_i is not $-$. The *size* of a rule $(\mathbf{t}, c)$, written $\mathrm{siz}(\mathbf{t}, c)$, is one plus the number of 0's and 1's in $\mathbf{t}$.

A *rule list* is a list of rules $(\mathbf{t}_1, c_1) \cdots (\mathbf{t}_k, c_k)$, with $\mathbf{t}_k = \mathbf{true}$ (the vector of all $-$'s). It represents the hypothesis h such that $h(\mathbf{x}) = c_i$, where $(\mathbf{t}_i, c_i)$ is the first rule that $\mathbf{x}$ matches. The *size* of a rule list is the sum of the sizes of its rules, i.e. the number of rules plus the total number of literals appearing in rules.

BBG takes as input a collection of examples called the *training sample*, which are presumed to be drawn independently and randomly from some unknown probability distribution $\mathcal{D}$ over $X \times C$. Note that this implies that the relationship between a vector $\mathbf{x}$ and its class c is in general a stochastic one, and not necessarily deterministic.

The *error* of a hypothesis h is the probability that $h(\mathbf{x}) \neq c$ when $(\mathbf{x}, c)$ is drawn at random from the same distribution $\mathcal{D}$ from which the training sample was drawn. The *empirical error* of a hypothesis h 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.

2 Outline of the Algorithm

The BBG algorithm consists of two parts:

1. An algorithm G which produces a sequence of rule lists of increasing size and decreasing empirical error.
2. A method for trading off empirical error and complexity to choose one of the rule lists output by G. (There is a tendency to fit noise or statistical flukes of the data when excessively complex hypotheses are allowed.[1,8])

These two parts are entirely independent. The literature contains various methods for the second part: cross-validation,[2] hold-out sets,[3] the minimum description-length principle,[6] structural risk minimization,[8] and heuristic error estimates.[5] BBG uses a heuristic error estimate described in a separate paper.[7] This paper concentrates on algorithm G, which is outlined below:

```
$i := 0$
$h_0 := h :=$ the single rule $(\textbf{true}, c)$, where $c$ is the most common class in $S$
while $(\mathrm{err}(h, S) > 0)$
  $(\mathbf{t}, c, p) := \mathrm{bestRule}(h, S)$
  insert rule $(\mathbf{t}, c)$ into $h$ just before its $p$-th rule
  $i := i + 1$; $h_i := h$
```

S is the training sample, and $h_0, h_1, \ldots$ is the output. The procedure bestRule() tries to find a rule $(\mathbf{t}, c)$ and position p maximizing the ratio (decrease in empirical error) / (rule size). We call this the *gain-cost ratio* of $(\mathbf{t}, c, p)$.

3 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 $\phi(y)$, for some given function ϕ. When a node z is reached one computes an upper bound u on the value of ϕ for leaf nodes descended from z; if u is no greater than $\phi(y^\star)$ for the best solution $y^\star$ found so far, there is no need to explore the subtree rooted at z.

3.1 Preliminaries

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

If $z = (\mathbf{v}, c, p)$ is a leaf node then $(\mathbf{v}, c)$ is a rule. If z is not a leaf node but $y = (\mathbf{t}, c, p)$ *is* a leaf node, then y is a descendant of z in the search tree if and only if $\mathbf{t}$ may be obtained from $\mathbf{v}$ 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. We define $z^\circ = (\mathbf{t}, c, p)$, where $\mathbf{t}$ is obtained from $\mathbf{v}$ by replacing each $*$ entry with –. We write solns$^\circ(z)$ for solns$(z) - \{z^\circ\}$. As will be seen, considering z° separately 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), $g^\star$ (the gain-cost ratio of the best solution found so far), and $y^\star$ (the best solution found so far).

3.2 The Branch-and-bound Algorithm

Letting $\ast = (*, \ldots, *)$, here is the algorithm for bestRule(h, S):

```
$Z := \emptyset$; $N := 0$; $g^\star := 0$
initialize information needed by eval(), using $h$ and $S$
for (each class $c$ and position $p$)
    $z := (\ast, c, p)$
    $(u, j) :=$ eval($z$)
    insert($u, j, z, Z$)
while ($Z \neq \emptyset$ and $N < \tau$)
    $(u, j, z) :=$ remove from $Z$ its best element
    while ($u > g^\star$)
        let $(\mathbf{v}, c, p) = z$
        $(z_0, z_1, z_2) :=$ children of $z$ obtained by setting $v_j$ to –, 1, and 0
        for ($i := 0$ to 2) $(u_i, j_i) :=$ eval($z_i$)
        if ($u_1 > u_2$) swap $(u_1, j_1, z_1)$ and $(u_2, j_2, z_2)$
        for ($i := 0$ to 1) insert($u_i, j_i, z_i, Z$)
        $(u, j, z) := (u_2, j_2, z_2)$
return $y^\star$
```

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').

The procedure eval(z) 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$^\circ(z)$ (see Section 3.3). 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: 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.

Note that bestRule() uses a combination of best-first search and depth-first search: it repeatedly removes the most promising node from Z and explores a linear path down from the node. As each node z is reached the solution z° is considered and evaluated. No leaf is ever expanded because the upper bound $u = 0$ associated with it is never more than $g^\star$.

3.3 Computing the Upper Bound

We now give the details of steps 2 and 3 of eval(). Let us define the following:

- $\Delta(y) = \mathrm{err}(h, S) - \mathrm{err}(h', S)$, where $y = (\mathbf{t}, c, p)$ is a leaf and h' is obtained from h by inserting the rule $(\mathbf{t}, c)$ at position p.
- $\mathrm{msiz}(z) = \mathrm{siz}(\mathbf{t}, c)$, where $z^\circ = (\mathbf{t}, c, p)$. Note that $\mathrm{msiz}(z) \leq \mathrm{msiz}(y)$ for any $y \in \mathrm{solns}(z)$.

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

Let us consider the computation of $\Delta(\mathbf{t}, c, p)$. We define the following:

- S_p = those examples in S that match none of the first $p - 1$ rules of h.
- $S_p^{\mathbf{t}}$ = those examples in S_p that match $\mathbf{t}$.
- $G(\mathbf{t}, c, p)$ = those examples $(\mathbf{x}, c')$ in $S_p^{\mathbf{t}}$ for which $c' = c$ and $h(\mathbf{x}) \neq c'$; $G(z) = G(z^\circ)$ if z is a non-leaf node.
- $B(\mathbf{t}, c, p)$ = those examples $(\mathbf{x}, c')$ in $S_p^{\mathbf{t}}$ for which $c' \neq c$ and $h(\mathbf{x}) = c'$.

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

Let $y = (\mathbf{t}', c, p) \in \mathrm{solns}^\circ(z)$ and $z^\circ = (\mathbf{t}, c, p)$. Then $S_p^{\mathbf{t}'} \subseteq S_p^{\mathbf{t}}$ (since $t_i' = t_i$ whenever t_i is not $-$), hence $G(y) \subseteq G(z^\circ) = G(z)$. Then $|G(z)| \geq |G(y)| \geq \Delta(y)$ for every $y \in \mathrm{solns}^\circ(z)$, and we obtain $|G(z)|/\mathrm{msiz}(z)$ as an upper bound on the gain-cost ratio for elements of $\mathrm{solns}^\circ(z)$.

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 $K[i] = h(\mathbf{x})$, where $(\mathbf{x}, c')$ is the i-th example. Letting $z = (\mathbf{v}, c, p)$ and $z^\circ = (\mathbf{t}, c, p)$, $|G(z)|$ is computed by stepping through the examples $S[f[p]]$ through $S[m]$, checking to see if each example $S[i] = (\mathbf{x}, c')$ matches $\mathbf{t}$, 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. In $\Theta(1)$ additional 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 msiz(z°) gives the gain-cost ratio for z°.

Eval() also use counters $\kappa[i,b]$ ($1 \le i \le n$, $b \in \{0,1\}$) initialized to 0. Once an example $(\mathbf{x}, c')$ has been identified as an element of $G(z)$, eval() uses $\Theta(n)$ time to step through the entries of $\mathbf{x}$, incrementing $\kappa[i, x_i]$ whenever $v_i = *$. This does not increase the asymptotic time complexity. Let $z[i:b]$ be the possible child of z obtained by replacing v_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 $v_i = *$. Since msiz$(z[i:b]) =$ 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 solns$(z[i:b])$. Taking the maximum over $b \in \{0,1\}$ and all i for which $v_i = *$ gives an upper bound on the gain-cost ratio for elements of 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 the following algorithm for steps 2 and 3 of eval():

```
let $(\mathbf{v}, c, p) = z$ and $(\mathbf{t}, c, p) = z^\circ$
$\gamma := 0$; $\beta := 0$; $d := 0$
init all $\kappa[i,b]$ to 0
for ($j := f[p]$ to $m$) if ($S[j]$ matches $\mathbf{t}$)
    let $(\mathbf{x}, c') = S[j]$
    if ($c' \ne c$ and $c' = K[j]$) increment $\beta$          — $S[j] \in B(z^\circ)$
    if ($c' = c$ and $c' \ne K[j]$)                            — $S[j] \in G(z^\circ)$
        increment $\gamma$
        for ($i := 1$ to $n$) if ($v_i = *$) increment $\kappa[i, x_i]$
$g := (\gamma - \beta)/\text{msiz}(z^\circ)$                   — gain-cost ratio of $z^\circ$
for ($i := 1$ to $n$ and $b := 0$ to 1) if ($\kappa[i,b] > d$)
    $d := \kappa[i,b]$; $j := i$
$u := d/(\text{msiz}(z) + 1)$
```

4 Experimental Results

We tested BBG on 1050 synthetic data sets. 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[5] 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. Each data set consisted of 500 training examples and 30000 test examples for estimating the actual error of the hypothesis output by a learning algorithm.

For BBG we set the memory limit μ and time limit τ as follows. Let m be the number of examples (500) and n the number of Boolean attributes. For the 500 data

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

l	r	n	k	η	BBG	c4.5	c4.5rules
29	12	38	3	0.03	4.85 $\pm$ 0.84	14.31 $\pm$ 1.65	13.15 $\pm$ 1.43
32	9	38	5	0.02	3.38 $\pm$ 0.51	15.58 $\pm$ 2.13	13.14 $\pm$ 1.70
30	11	45	5	0	0.45 $\pm$ 0.26	11.86 $\pm$ 1.64	8.75 $\pm$ 1.70
34	7	50	4	0.04	7.56 $\pm$ 0.77	20.78 $\pm$ 2.32	20.78 $\pm$ 2.14
34	7	52	2	0.05	9.05 $\pm$ 1.01	17.22 $\pm$ 1.50	16.49 $\pm$ 1.37
30	11	54	6	0	0.58 $\pm$ 0.30	13.15 $\pm$ 1.84	9.88 $\pm$ 1.57
32	9	61	4	0.02	4.19 $\pm$ 0.82	16.69 $\pm$ 1.69	15.18 $\pm$ 1.71
36	5	64	3	0.05	11.21 $\pm$ 0.93	17.96 $\pm$ 1.30	19.38 $\pm$ 1.29
31	10	67	3	0.04	6.97 $\pm$ 0.80	20.70 $\pm$ 2.06	19.07 $\pm$ 2.04
28	13	79	2	0.02	6.36 $\pm$ 1.02	12.75 $\pm$ 1.63	11.66 $\pm$ 1.52

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 $\max(250000, 10mn)$. Typical execution times on an HP 710 workstation were around 25 minutes.

The data sets were produced using two problem generators, rgenex and tgenex. These generators take the parameters n, k (number of classes), η (noise level), m, and m'. Rgenex takes additional parameters r and l, and tgenex takes the additional parameter s. Rgenex and tgenex work as follows:

1. A *target* h is randomly generated. For rgenex h is a rule list having r non-default rules and size $r + l + 1$. In addition, each class c appears in approximately $(r + 1)/k$ rules and h has no superfluous rules (rules matched only when some preceding rule is also matched.) For tgenex h is a decision tree having s leaf nodes, with no class labeling more than twice as many leaf nodes as any other class. In both cases we have n binary attributes and k classes.
2. A probability distribution $\mathcal{D}$ over $\{0, 1\}^n$ is constructed. For rgenex each of the rules is equally often the first matched, and for tgenex the s leaf nodes are equally often reached. The distribution "within" each rule / leaf is uniform.
3. m training examples and m' testing examples $(\mathbf{x}, c)$ are generated. First, $\mathbf{x}$ is randomly selected from $\mathcal{D}$. Then with probability $1 - \eta$ we set $c = h(\mathbf{x})$, and with probability η we set $c = d$ for some class $d \neq h(\mathbf{x})$ chosen uniformly and randomly. This gives a uniform noise level of η.

Complete details on rgenex and tgenex are in another paper.[7]

For any fixed setting of the parameters to rgenex and tgenex one can investigate the expected error of a learning algorithm by generating a series of data sets using these parameters, running the algorithm on the training samples and averaging the resultant error rates on the test samples. We generated 50 data sets for each parameter setting investigated, in order to get a good estimate of the expected error rate, and also calculated 95% confidence intervals. Note that the target h has an error of η, which is the minimum possible error.

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

s	n	k	η	BBG	c4.5	c4.5rules
20	38	3	0.03	5.33 $\pm$ 0.56	14.27 $\pm$ 1.93	8.58 $\pm$ 1.23
20	38	5	0.02	2.69 $\pm$ 0.28	5.56 $\pm$ 1.08	3.56 $\pm$ 0.60
20	45	5	0	0.58 $\pm$ 0.32	4.46 $\pm$ 1.28	0.67 $\pm$ 0.22
20	50	4	0.04	5.43 $\pm$ 0.49	11.76 $\pm$ 1.64	8.56 $\pm$ 1.00
20	52	2	0.05	15.69 $\pm$ 1.90	30.14 $\pm$ 2.13	28.53 $\pm$ 2.44
20	54	6	0	0.76 $\pm$ 0.43	2.39 $\pm$ 0.77	0.59 $\pm$ 0.18
20	61	4	0.02	3.77 $\pm$ 0.54	10.78 $\pm$ 2.18	5.91 $\pm$ 1.35
20	64	3	0.05	9.68 $\pm$ 0.96	22.61 $\pm$ 2.86	18.08 $\pm$ 2.58
20	67	3	0.04	8.66 $\pm$ 1.36	20.87 $\pm$ 2.49	14.90 $\pm$ 1.98
20	79	2	0.02	14.95 $\pm$ 2.63	30.74 $\pm$ 2.43	27.93 $\pm$ 3.03
20	50	2	0	3.33 $\pm$ 0.93	24.64 $\pm$ 2.34	18.04 $\pm$ 2.59

We first compared BBG and C4.5 for 10 different parameter settings of rgenex randomly chosen subject to $r + l = 41$. The results are summarized in Table 1, where the columns labeled BBG, c4.5 and c4.5rules give the average error of the three learning algorithms (in percent). For each of the ten settings of parameter values, BBG outperforms both c4.5 and c4.5rules by wide margins.

We next compared BBG and C4.5 for 11 different parameter settings of tgenex. We set $s = 20$ in all cases and, except for the eleventh parameter setting, kept the same values for n, k, and η as used for rgenex.

The results are summarized in Table 2. 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).

Next we compared BBG and C4.5 on four data sets from the UC Irvine machine learning repository:[4] chess, tic-tac-toe, mushroom, and zoo. All of these have only nominal attributes. We made a minor modification to BBG to handle non-binary attributes, described elsewhere.[7] We handled unknown attribute values by having BBG treat "unknown" as just another possible attribute value.

We randomly partitioned the mushroom data set into 200 training examples and 7924 test examples (the problem is too easy if we use more training examples). Each of the other data sets was randomly partitioned into 10 nearly-equal parts and the learning algorithms run 10 times, each time choosing one part as the test sample and the remaining data as the training sample; the errors were averaged. Table 3 contains the results, with m, n and k having the same meaning as before, and error rates in percent. BBG used $\mu = 50000$ and $\tau = 250000$ in all cases.

Though not bad, these last results are a bit disappointing. This may simply be due to chance, given the small number of data sets, but there is another possibility. Our analysis of bestRule() reveals that that if there are strong correlations between

Table 3: Comparison of BBG and C4.5 on some UCI repository data sets

Data Set	m	n	k	BBG	c4.5	c4.5rules
chess	3196	36	2	0.97	0.47	0.44
tic-tac-toe	958	9	2	1.15	14.61	1.04
mushroom	200	22	2	1.72	1.49	1.11
zoo	90	16	7	6.67	10.0	10.0

attributes, *and these correlations are independent of class*, BBG's branch-and-bound search could be led down unproductive paths. While such correlations would be uncommon in our synthetic data sets, they might be much more common in real-world data sets.

5 Conclusions and Directions for Further Research

Our results indicate considerable promise for BBG, although the matter of correlated attributes needs to be addressed; this can be investigated with more sophisticated problem generators. Further research will look at other implementations of bestRule() intended to be more robust and able to handle real-valued attributes and internal disjunction for nominal attributes.

References

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

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

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

[4] P. M. Murphy & D. W. Aha (1992.) *UCI Repository of machine learning databases.* Irvine, CA: University of California at Irvine, Dept. of Information and Computer Sciences.

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

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

[7] K. S. Van Horn & T. R. Martinez (1993.) The Design and Evaluation of a Rule Induction Algorithm. Technical Report BYU-CS-93-11, Computer Science Department, Brigham Young University.

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