Approximation Via Value Unification[1]

Paul E. Utgoff utgoff@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

David J.. Sracuzzi stracudj@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

Abstract: Numerical function approximation over a Boolean domain is a classical problem with wide application to data modeling tasks and various forms of learning. A great many function approximation algorithms have been devised over the years. Because the goal is to produce an approximating function that has low expected error, algorithms are typically guided by error reduction. This guiding force, to reduce error, can bias the algorithm in a detrimental manner. We illustrate this bias, and then propose an alternative approach based on a notion of value unification.

1 Introduction

Suppose that we are given a batch of N labeled training points, each of the form $(\mathbf{b}, v(\mathbf{b}))$. The point $\mathbf{b}$ is represented as a vector of d Boolean variables. The real value $v(\mathbf{b})$ is the target value for point $\mathbf{b}$, which means that it is the ideal value for the approximator to return, noise in the measurement of $v()$ notwithstanding. Our problem is to find a good approximation $\hat{v}$ such that the desired error measure is minimized. A commonly used error measure is mean-squared error $\frac{1}{N}\sum_{j=1}^{N}(\hat{v}(\mathbf{b}_j) - v(\mathbf{b}_j))^2$. Furthermore, we desire a $\hat{v}$ that has low error for any set of $(\mathbf{b}, v(\mathbf{b}))$ points that could be drawn from the same distribution as the training points.

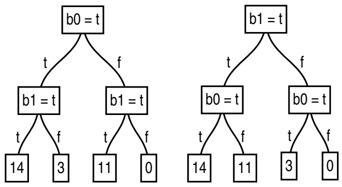
It is common to map non-Boolean variables to Boolean. For example, a discrete variable of k values can be mapped to k Boolean variables. For a given discrete value, the corresponding Boolean variable is set to `true`, and the others for that discrete variable are set to `false`. Similarly, a continuous variable can be partitioned into k intervals, and hence k Boolean variables. For a given real value, the Boolean variable corresponding to the interval in which the value falls is set to `true`, and the others for that continuous variable are set to `false`. We do not revisit these mapping procedures here, and consider the problem in Boolean form.

2 Regression Tree Induction as Approximation

One method of function approximation is to form a tree-structured regression, known as a *regression tree* (Breiman, Friedman, Olshen & Stone, 1984). The domain $\mathbf{b}$ is partitioned recursively into a finite set of blocks, and for each block a constant value, e.g. the mean, is fixed as the value of the approximation $\hat{v}$ for every point in that block. The partition represents a piece-wise constant function.

How is the partition determined? A common approach is to split any block in which the variance of the $v()$ values therein is too large, as measured by exceeding a user specified threshold. Variance is equivalent to the mean squared error when approximating by the mean of the sample,

[1]The correct citation for this article is: Utgoff, P. E., & Stracuzzi, D. J. (1999). Approximation via value unification. *Proceedings of the Sixteenth International Conference on Machine Learning* (pp. 425-432). Ljubljana: Morgan Kaufmann.



(a) (b)

Figure 1. The Two Possible Regression Trees

so this is the same as minimizing error. A split is accomplished by enumerating the possible tests, and selecting a test for which the average error in the resulting blocks would be minimized. For our purposes, each Boolean variable b_i can serve as a binary test. Let L be the instances in one block, and R be the instances in the other block. Then the assessment of the test is $\frac{|L|\sigma_L^2+|R|\sigma_R^2}{|L|+|R|}$. A test (Boolean variable) with the lowest value of this metric is selected and installed at the decision node, and each of the two resulting blocks is partitioned recursively until each block is of sufficiently low error (variance). In the extreme, every block will contain a single training point, but typically there is a useful amount of generalization, producing fuller blocks and a smaller tree.

3 Bias of Error Reduction

Consider a simple approximation task for instances of two Boolean variables. Suppose that the target function is $v(\mathbf{b}) = 3b_0 + 11b_1$, and that we have been given all four possible instances for training. Because $v()$ does not include any second degree terms, these two Boolean variables make independent contributions to the value of the function. We should not expect either variable to be preferred for the purpose of building a regression tree. However, a regression tree inducer guided by error (variance) reduction will prefer to test b_1 at the root. As shown in Figure 1, testing b_0 at the root groups 14 with 3 in one block, and 11 with 0 in the other. However, testing b_1 at the root groups 14 with 11 in one block, and 3 with 0 in the other, which produces lower expected error. So, although we happen to know in this case that either test is equally good, we see that the test selection metric is biased to favor one over the other. This bias is misleading.

4 Value Unification

Let us examine this same simple function approximation problem from a different point of view. Figure 2(a) depicts the target function v as a Venn diagram, where each value shown is the error with respect to $\hat{v}$, which is initially the 0 function. A set contains those instances for which the corresponding Boolean variable is `true`. Suppose that we could infer the $3b_0$ component of the target function. Then the error function would change to that of Figure 2(b). The point that is in the b_0 set but not the b_1 set now has error 0, and the error for the point that is in both the b_0 set and the b_1 set now has error -11. Originally, there were four different values, but now there are two. A unification of values has occurred, making the remaining approximation problem simpler. Notice that if we had instead first inferred the $11b_1$ component of the target function, a different but equally useful unification would have occurred.

These observations suggest that an induction process based on value unification is possible, and that it would be biased differently from one based on error reduction. The variance reduction bias first groups instances by target value, and then finds a test to facilitate that grouping. This can be very misleading because instances can have similar values for different reasons. In contrast, the

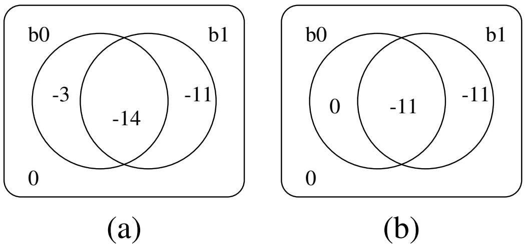


Figure 2. Unification of Values

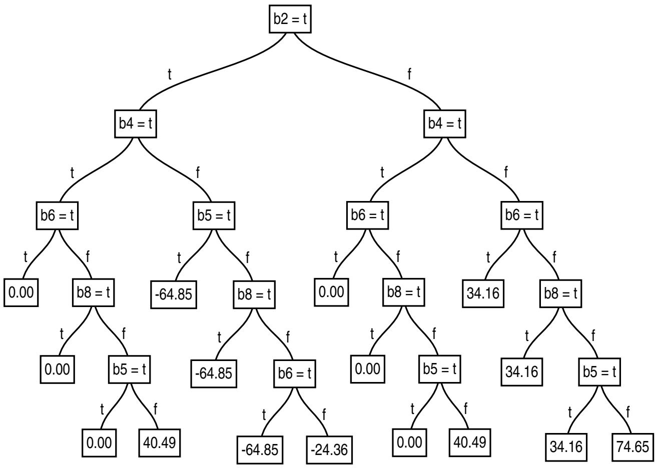


Figure 3. Regression Tree Using Variance Metric

unification bias seeks to account for the reasons that an instance has the value it does. For example, one instance may have value 20 because of evaluation $10b_7 + 10b_9$, whereas another instance may have value 20 because of different feature contributions $5b_3 + 9b_5 + 6b_8$. Attempting to group these two instances because they each evaluate to 20 is misguided.

5 Comparison of Biases in Regression Tree Induction

In this section we digress briefly from our quest to formulate a function approximation algorithm driven primarily by value unification instead of primarily by error reduction. For discussion purposes, we fashion a regression tree algorithm that is driven by value unification. Then we consider how different such a mechanism is from classification tree induction.

For regression tree induction we do not need to unify any values, but we can be guided by the same goal. By substituting a new test selection metric into the same splitting mechanism described above, we can reduce the total number of distinct values observed. Let $u(T)$ be the number of distinct $f(\mathbf{b})$ values in T, where T is a set of training points. Define the test selection metric to be $u(L) + u(R)$, where as before L and R are the two blocks of the partition. A test is good to the extent that it separates instances into blocks with small value sets. Note that for regression tree induction, it is not necessary to unify the values of the blocks since they are handled independently, but the motivation is the same.

It is instructive to examine two regression trees for a single small problem. Figure 3 shows a typical example in which partitioning based on error reduction, called *error_tree*, does a poor job of

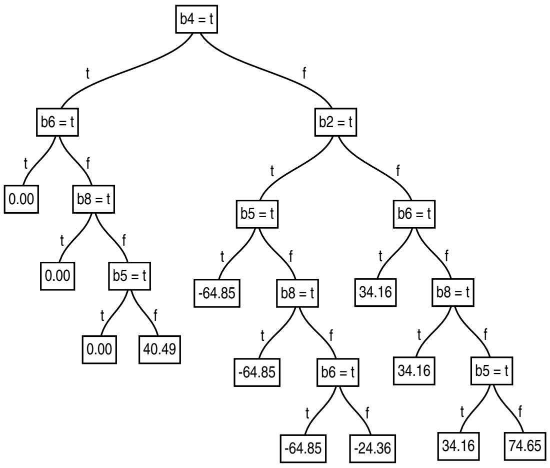


Figure 4. Regression Tree Using Unification Metric

grouping values, and hence capturing the regularity in the data. Notice that error_tree sends four of six values down the left branch and another four of six values to the right, causing much duplication of subproblems and replication of subtrees. Figure 4 shows the tree built for the same problem using the unification metric, called *unify_tree*. Two of the six possible values are sent down the left branch, and the remaining four are sent down the right branch, eliminating duplication in the subproblems. The FRINGE algorithm (Pagallo & Haussler, 1990) would be able to shrink either of these trees by constructing conjunctive tests. Nevertheless, one should not count on FRINGE to overcome a bad bias in every case.

How do these two test selection metrics compare in practice? We could measure error on independent training and testing sets, but here we simply look at the number of leaves, assuming that a smaller tree does a better job of finding the regularity in the data. Table 1 shows tree size for four different algorithms under several combinations of 'number of binary variables' and 'number of features in the target function'. Two classification tree inducers ITI (Utgoff, Berkman & Clouse, 1997) and C4.5 (Quinlan, 1993) are included. For those, the numeric target value for each instance is treated as a non-numeric class label. The tree sizes shown in the table are computed by averaging the number of leaves for 100 artificial problems.

Based on these runs, unify_tree builds smaller trees than error_tree. This suggests that grouping instances by numeric closeness of target value does not offer an advantage, and may be detrimental. ITI finds still smaller trees, suggesting that its test selection metric, a form of Kolmogorov-Smirnoff distance (Friedman, 1977; Utgoff & Clouse, 1996), is better. It might be fruitful to pursue this digression a little further, but the purpose here was to illustrate that an error reduction approach to regression tree induction has a bias, and that it is not favorable to the induction processes, as measured by tree size.

6 Feature Construction Guided by Value Unification

Let us return to the problem of designing an algorithm that fits the set of N training points. The algorithm is to be guided primarily by unification of values. To motivate the algorithm that is presented below, consider first what happens when a feature and its coefficient are appended to a function. This provides a sense of how the myriad values come into being, which will be useful for trying to reverse the process. Given the data points, we will want to be able to determine a set of features and coefficients that would produce them.

Define the function approximator of no features to have value 0 for all points, giving $v(\mathbf{b}) = 0$. This function has a one element value set $\{0\}$. Now define a feature $f_0(\mathbf{b})$ to evaluate to 1 for any point in which variable b_0 is `true`, and to evaluate to 0 otherwise. Define c_0, $c_0 \neq 0$, to be its coefficient, giving $v(\mathbf{b}) = c_0 f_0(\mathbf{b})$. This function v has a two element value set $\{0,\ c_0\}$. The data points covered by feature $f_0(\mathbf{b})$ evaluate to c_0, and the remaining points evaluate to 0. A feature makes a non-zero contribution to the value of each instance that it covers.

For the sake of discussion, and following our earlier example, let $c_0 = 3$. Add a new feature $f_1(\mathbf{b})$ that covers exactly those points for which variable b_1 is `true`, and define its coefficient $c_1 = 11$. Our function is now $v(\mathbf{b}) = 3f_0(\mathbf{b}) + 11f_1(\mathbf{b})$, which has value set $\{0,\ 3,\ 11,\ 14\}$. An instance that is covered by a feature has a non-zero value. That is not true in general, because the sum of a set of coefficients could be zero, but the discussion will be simpler if for now we assume that no covered instance evaluates to 0.

6.1 Top Level

We can construct an algorithm that attempts to reproduce the features and their coefficients that define the target function v. The reproduction $\hat{v}$ may not be exact, making it an approximator of v. Our approach, embodied in an algorithm called *VU*, is to identify the coefficient-feature pairs sequentially, searching in the general-to-specific direction in the space of features, as defined by the sets of instances covered. For example, the most general feature is the one that covers all possible instances in instance space.

Consider the top level of the algorithm:

1. Receive N training points $(\mathbf{b}, v(\mathbf{b}))$ in d variables.
2. For each training point, initialize its error to $0 - v$.
3. Initialize $\hat{v} = 0$.
4. While some instance has non-zero error:
   (a) Find the next feature.
   (b) Find the feature's coefficient.
   (c) Append the term to the definition of $\hat{v}$.
   (d) For each instance covered by the new feature, add the coefficient to its error.

This looks very much like an error-reduction procedure, and it is. To unify values is to reduce error. However, the algorithm is not grouping instances by size of error, nor is it trying to reduce error for all instances simultaneously. Instead, it is unifying values by extracting coefficient-feature pairs one at a time. When one does this successfully, the size of the value set of the adjusted instance values is reduced. Ultimately, all adjusted instance values (the instance error values) reach 0.

Watch how the algorithm works for the following training instances of two variables. The operations of finding the next feature and its coefficient are treated as black boxes for the moment.

b_0	b_1	$v(\mathbf{b})$	error
1	1	14	-14
1	0	3	-3
0	1	11	-11
0	0	0	0

At this point, we have $\hat{v}(\mathbf{b}) = 0$. Now the feature $f_0(\mathbf{b})$ and its coefficient 3 are found. The new term is added to the approximator, giving $\hat{v}(\mathbf{b}) = 3f_0(\mathbf{b})$, and the error values are adjusted, giving:

b_0	b_1	$v(\mathbf{b})$	error
1	1	14	-11
1	0	3	0
0	1	11	-11
0	0	0	0

The unification of values is taking place on the error values. The size of the value set has been reduced from 4 to 2. One more iteration of the top level would finish the process by unifying all error values to 0, with $\hat{v}(\mathbf{b}) = 3f_0(\mathbf{b}) + 11f_1(\mathbf{b})$.

6.2 Finding A Feature

The procedure for finding the next most general feature is implemented as best-first search, but in practice it amounts to hill-climbing. The `open` list is initialized with the most general feature, which covers all instances. A feature that covers an instance with error (unified) 0 can be specialized. Indeed, that is desirable because it means that a more specialized feature will not alter the value of an instance that currently has error 0.

The metric for the best-first search is the number of training instances covered by the feature that have error 0, and the search prefers nodes with lower values of the metric. The search terminates when any of three conditions is met:

1. the fewest-true-values instance (`true` values for exactly those variables tested in the feature cover definition) is present in the training instances,

2. the metric value for the best feature on the `open` list is 0 (no instances with 0 error are covered),

3. the metric value for the best feature on the `closed` list is not greater than the best feature on the `open` list (no progress in minizing the number of covered 0 error instances).

Until the search terminates, the best feature is removed from `open` and placed onto `closed`, and the successors of that currently best feature are inserted into the `open` list. A successor is generated by copying the feature definition, and then conjoining a missing variable to it. For example, the most general feature is 1 because it evaluates to 1 regardless of the values of the variables in an instance. A specialization of the most general feature is b_0 because it evaluates to 1 for just those instances in which b_0 is `true`, and to 0 otherwise. If a successor feature covers no instances, it is discarded immediately without putting it onto the `open` list.

6.3 Finding a Feature's Coefficient

The VU algorithm finds features one at a time, and in general-to-specific order. For this reason, it is essential to pay attention to whether the fewest-true-values instance covered by the feature is present in the training instances. Consider the instance in which all variables are `false`.

No specialized feature can cover this instance. To choose a coefficient that does not unify this instance with 0 will produce a permanent error. So, in this case, the coefficient can be extracted directly from the training instance.

When the fewest-true-values instance for a feature is not present in the data, how does one find a coefficient? We could choose a value that would most reduce the size of the resulting value set. This would be most obviously in keeping with the principle of value unification. However, this measure is sometimes ambiguous. A second possibility is to choose a value that unifies the largest number of instances with 0. Either of these approaches works well, but neither is foolproof. A third approach follows from the reasoning above about wanting to handle fewest-true-values instances. Find a fewest-true-values training instance from those covered by the feature, and use its weight as the coefficient. This approach may not be foolproof either, but we have not yet observed it to make an error. It unifies the values as we would like, so it achieves our purpose.

6.4 Illustration

Here is a trace of the VU algorithm for a small approximation task in which there are $d = 8$ Boolean variables, giving $2^d = 256$ possible points $\mathbf{b}$. The target function consists of the three indicated features, making eight possible v values. The 100 training instances were drawn at random with replacement.

A feature definition is depicted as a string of '.'s and '1's. A dot means that the variable value is not relevant to the value of the feature, whereas a one means that it is relevant. The feature depicted by all dots is the most general feature. A feature evaluates to 0 if any of the tested variables (indicated by a '1' in the string) is `false`, and 1 otherwise. The `open` and `closed` lists are summarized by showing the metric value (number of training instances covered with 0 error) for up to the five best features on the list. The word 'quality' refers to metric value.

```
Using 8 variables
Using 3 features in target
Using 100 training instances

Target
   3.00 ....11..
   4.00 11......
   5.00 .1..1...

Closed:
Open:   61

Expand  ........
  Child .......1 quality    34
  Child ......1. quality    35
  Child .....1.. quality    16 <-- best
  Child ....1... quality    21
  Child ...1.... quality    33
  Child ..1..... quality    31
  Child .1...... quality    16 <-- best
  Child 1....... quality    21
```

```
Closed: 61
Open:   16 16 21 21 31 ...

Expand  .....1..
  Child .....1.1 quality     8
  Child .....11. quality    12
  Child ....11.. quality     0 <-- best
  Child ...1.1.. quality     6
  Child ..1..1.. quality    11
  Child .1...1.. quality     6
  Child 1....1.. quality     3

Closed: 16 61
Open:   0 3 6 6 8 ...

Weight 3.00 from hamming dist 0 instance
Append  ....11.. to approx with weight 3.00

Closed:
Open:   75

Expand  ........
  Child .......1 quality    39
  Child ......1. quality    42
  Child .....1.. quality    30
  Child ....1... quality    35
  Child ...1.... quality    40
  Child ..1..... quality    38
  Child .1...... quality    16 <-- best
  Child 1....... quality    25

Closed: 75
Open:   16 25 30 35 38 ...

Expand  .1......
  Child .1.....1 quality     8
  Child .1....1. quality     8
  Child .1...1.. quality     6
  Child .1..1... quality     0 <-- best
  Child .1.1.... quality     5
  Child .11..... quality     7
  Child 11...... quality     0 <-- best

Closed: 16 75
Open:   0 0 5 6 7 ...

Weight 5.00 from hamming dist 1 instance
```

```
Append  .1..1... to approx with weight 5.00

Closed:
Open:   82

Expand  ........
  Child .......1 quality    40
  Child ......1. quality    45
  Child .....1.. quality    34
  Child ....1... quality    42
  Child ...1.... quality    41
  Child ..1..... quality    43
  Child .1...... quality    23 <-- best
  Child 1....... quality    25

Closed: 82
Open:   23 25 34 40 41 ...

Expand  .1......
  Child .1.....1 quality     9
  Child .1....1. quality    11
  Child .1...1.. quality    10
  Child .1..1... quality     7
  Child .1.1.... quality     6
  Child .11..... quality    12
  Child 11...... quality     0 <-- best

Closed: 23 82
Open:   0 6 7 9 10 ...

Weight 4.00 from hamming dist 0 instance
Append  11...... to approx with weight 4.00

Approximator:
   4.00 11......
   5.00 .1..1...
   3.00 ....11..
```

6.5 Sample Size for Training

How many training instances are needed for the VU algorithm? One must sample the instance space adequately so that the features can be extracted from the data. The VU approach to feature construction depends on the ability to discriminate instances that should be covered by a feature from those that should not. Thus the problem of finding a feature is one of concept learning, and the algorithm for finding a feature definition conducts a general-to-specific search (Mitchell, 1982), with non-zero values denoting positive instances, and zero-values denoting negative instances. Concept learning occurs for each feature, and is embedded in the larger VU process of inducing a function approximator. Because the features are found one at a time, we have a sequence of concept learning problems. Upon learning finding a feature definition (learning a concept), the outer process finds a coefficient and adjusts the instance values.

We can apply sample complexity results for concept learning to the VU approach to approximator induction. We have here a sequence of concepts to be learned, so a small extension is required. Ehrenfeucht, Haussler, Kearns, & Valiant (1988) provide a lower bound m on the sample size that is based on three parameters: the size of the hypothesis space $|H|$, the acceptable misclassification rate ε, and the probability δ of failing to induce a concept with misclassification rate at most ε. For a target function of one feature, the size of the hypothesis space $|H|$ is 2^d, where d is the number of Boolean variables. The lower bound can be computed directly. For two features, we have two concept learning problems. One might say that the one original sample will suffice for each problem, but it is important to keep in mind that one is rolling a $\frac{1}{\delta}$-sided die for each feature (concept) to be found. Because δ is the probability of failure, $1-\delta$ is the probability of success. We need a μ such that $(1-\mu)^k = (1-\delta)$, where k is the number of features in the target function. Rearrangement of terms gives $\mu = 1-(1-\delta)^{\frac{1}{k}}$. Use μ in place of δ when computing the sample size for the sequence of concept learning problems. The effect is to increase the sample size. We do not know ahead of time how many features are in the target function. We could attempt to estimate k, but it is fruitless to consider a uniform distribution of possible functions to be approximated (Rao, Gordon & Spears, 1995).

The VU algorithm fails when the training instances are not sufficient to enable it to paint the feature boundaries. Our remedy is to increase the sample size, but we would like to make the algorithm robust to this condition.

6.6 Feature Cancellation

The VU algorithm will also fail when an instance has 0 error because it is covered by features whose coefficients sum to 0. The best-first search for a feature that excludes all instances with error 0 eventually degenerates into producing features that cover individual instances. Work is underway to remove this limitation.

6.7 Computational Efficiency

The algorithm can be implemented efficiently. The number of instances that are covered by a feature and have 0 error can be computed in a single sweep over the instances. In terms of sweeps over the instances, the VU algorithm requires $O(d^2)$ sweeps per feature to be found.

6.8 Empirical Evaluation

We have only tested this particular algorithm on artificial target functions of the same form used by the approximator itself. More work is needed to see whether other kinds of targets have any important differences. We have varied the value of d, the distribution of coefficient values for the target features, the distribution over the number of bits to set in a features, the number of features in the target concept, and the number of training instances. We have not tested the algorithm on 'real' problems, nor have we investigated issues relating to noise and overfitting.

Regarding noise, we expect that small amounts of noise in the function value can be tolerated by relaxing the test for unified values. Instead of requiring true equality, we can treat similar values as equal. For instances that are outliers by virtue of large errors, either in the variable values or the function value, we expect that the algorithm would construct a very specialized feature to cover each outlying instance. Such features could be discarded afterward, constituting a form of pruning.

We have not done any experimentation regarding noise, so these thoughts are speculative.

6.9 Other Approximator Inducers

A great many methods have been devised for approximating functions, most of which are guided primarily by error minimization. Value unification provides a different avenue for finding regularity in the data. We decided to test an artificial neural network inducer (Rumelhart & McClelland, 1986) on the target function shown in Table 2. The target function has $d = 20$ Boolean variables and $k = 10$ features, and $10,000$ training instances were generated at random with replacement. The artificial neural network had 20 input units, 10 sigmoid threshold hidden units, and 1 unthresholded output unit. Such an architecture is sufficient for representing the target function because a hidden unit can represent a set cover feature, and the output unit can represent a linear combination of the feature values.

Counting a network output as correct if it is within 5% of the target output, and letting backprop run for 320 epochs produces a network that is 'correct' for 9,501 of the 10,000 examples. Allowing the training to continue for 20,000 epochs improves accuracy only slightly. Each 100 epochs takes about 1 minute of cpu. For some purposes, 95% accuracy may be sufficient, but it is clear that the network is not finding all the regularity in the data. In contrast, VU finds an exact solution in 9 cpu seconds.

This is just one problem, but it is another kind of indication that one may not find regularity when striving too hard to reduce all error at one time. A more systematic set of comparisons is needed.

6.10 Compactness of Overlapping Features

The VU algorithm produces a compact representation in the form of overlapping features. This offers considerable economy compared to a regression tree. For example, for the target function in Table 2, a regression tree based on the unification metric contains 879 leaves, versus 10 features for VU. The regression tree algorithm (unify_tree) requires 15 seconds to construct the tree, versus 9 seconds for VU to find its approximation. Regression trees based on the input variables, whatever the metric, partition the space into disjoint regions.

7 Summary

Our work presents three ideas. The first is that algorithms that are driven primarily by error reduction can be led into missing regularities in the data that would facilitate function approximation. Second, it is beneficial to try to group instances that have similar values for similar reasons. Finally, it is possible to fashion algorithms that are driven primarily by value unification, and only secondarily by error reduction. In particular we presented the novel VU algorithm that uses value unification to identify useful overlapping features.

Acknowledgments

This work was supported by National Science Foundation Grant IRI-9711239. We are indebted to Gang Ding, who produced the neural network results. We thank David Jensen, Doina Precup, Amy McGovern, and Yanfeng Lu for providing helpful comments.

References

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Belmont, CA: Wadsworth International Group.

Ehrenfeucht, A., Haussler, D., Kearns, M., & Valiant, L. (1988). A general lower bound on the number of examples needed for learning. *Proceedings of the 1988 Workshop on Computational Learning Theory* (pp. 110-120). San Mateo, CA: Morgan Kaufmann.

Friedman, J. H. (1977). A recursive partitioning decision rule for nonparametric classification. *IEEE Transactions on Computers, C-26*, 404-408.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18*, 203-226.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning, 5*, 71-99.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. San Mateo, CA: Morgan Kaufmann.

Rao, R.B., Gordon, D. , & Spears, W. (1995). For every generalization action, is there really an equal and opposite reaction? Analysis of the Conservation Law for Generalization Performance. *Machine Learning: Proceedings of the Twelfth International Conference* (pp. 115-121). Tahoe City, CA: Morgan Kaufmann.

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing*. Cambridge, MA: MIT Press.

Utgoff, P. E., Berkman, N. C., & Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning, 29*, 5-44.

Utgoff, P. E., & Clouse, J. A. (1996). *A Kolmogorov-Smirnoff metric for decision tree induction*, (Technical Report 96-3), Amherst, MA: University of Massachusetts, Department of Computer Science.

#vars	#features	unify_tree	error_tree	iti	c4.5
10	10	151.77	165.71	145.88	290.68
10	20	557.82	626.68	553.43	1105.27
15	10	608.60	654.90	519.76	968.07
20	10	1310.05	1497.64	1167.71	1833.42
25	10	2204.29	2617.21	2040.40	2696.58

Table 1. Tree Induction Measurements

Table 2. Target Function

```
Target:
   5.00 .1.....1.....1..11..
   3.00 .......11.11.....1..
   2.00 .11........1.......1
   4.00 ....1.......1..11...
   1.00 .1..............11..
   5.00 ..........1......11.
   1.00 .....1.......1......
   4.00 ...............1..1.
   3.00 ....1............1..
   5.00 ...........1.1......
```