Approximation Via Value Unification

Paul E. Utgoff
David J. Stracuzzi

Technical Report 99-19
January 28, 1999

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Telephone: (413) 545-4843
Net: utgoff@cs.umass.edu

Contents

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. We compare these two biases in the realm of regression tree induction, and then present a new function approximation algorithm AVVU based on value unification.

1 Introduction

Suppose that we are given a batch of N labeled training points, each of the form $(\mathbf{b}, f(\mathbf{b}))$. The point $\mathbf{b}$ is represented as a vector of d Boolean variables. The real value $f(\mathbf{b})$ 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 $f()$ notwithstanding. Our problem is to find a good approximation $\hat{f}$ such that the mean-squared error $\frac{1}{N}\sum_{j=1}^{N}(\hat{f}(\mathbf{b}_j) - f(\mathbf{b}_j))^2$ is minimized. Furthermore, one desires an $\hat{f}$ that has low error for any set of $(\mathbf{b}, f(\mathbf{b}))$ points that could be drawn from the same distribution as the training points.

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 $\hat{f}$ 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 $f()$ values therein is above some predetermined threshold. Variance is equivalent to the mean squared error when approximating by the mean of the sample, 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 variance 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 a decision node in the regression tree, 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 hopefully there will be good 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 $f(\mathbf{b}) = 3b_0 + 11b_1$, and that we have been given all four possible instances for training. Because $f()$ 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. 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 would rather that the test selection metric not be biased in this case, we see that it is nevertheless.

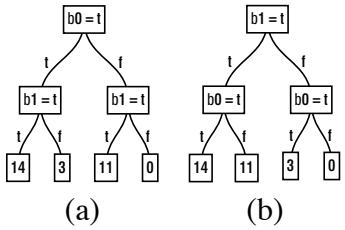

Figure 1. The Two Possible Regression Trees

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 f as a Venn diagram, where each value is the error with respect to $\hat{f}$, 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 in the b_0 set but not the b_1 set now has error 0, and the error for the point in 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 occured, 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.

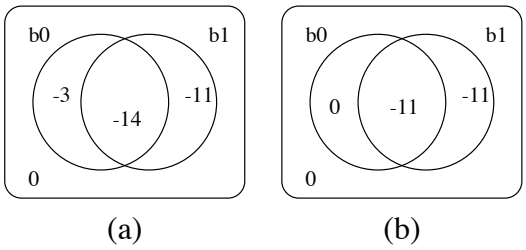

Figure 2. Unification of Values

These observations suggest that an induction process based on value unification is possible, and that it would be biased differently from error reduction. The variance reduction bias first groups instances by target value, and then finds a test to facilitate that desired grouping. The unification bias seeks to account for the differences in instance values with respect to each other, instead of with respect to the target values. Of course the final approximator has the same goal in each case, to reduce error, but the priority for grouping instances is different.

We shall show first how a bias based on value unification can be manifested in a regression tree inducer. Afterward, we present a new algorithm AVVU of a different kind.

5 Comparison of Biases in Regression 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 $v(\mathbf{T})$ be the number of distinct $f(\mathbf{b})$ values in $\mathbf{T}$, where $\mathbf{T}$ is a set of training points. Define the test selection metric to be $|v(L)| + |v(R)|$, 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.

#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

How do these two test selection metrics compare in practice? One question is which algorithm builds smaller trees? Table 1 shows a variety of measurements that give some indication. For each combination of number of binary variables and number of features in the target function, 100 artificial problems were given to unify_tree (induce regression tree using unification metric) and error_tree (induce regression tree using expected error metric). The average number of leaves for each set of 100 problems is shown in the table. Apparently the unify_tree algorithm builds smaller trees than error_tree.

This leads to a second question, whether one might reasonably employ a decision tree inducer that treats the target values of the instances as tokens, not as numerical values. The same problem sets were given to each of ITI (Utgoff, Berkman & Clouse, 1997) and C4.5 (Quinlan, 1993). The tree sizes for unify_tree and ITI are quite similar, with an apparent edge to ITI. We shall leave implications for tree-structured induction to another discussion.

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 does a poor job of 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 (Pagallo & Haussler, 1990). Figure 4 shows the tree built by unify_tree for the same problem. 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.

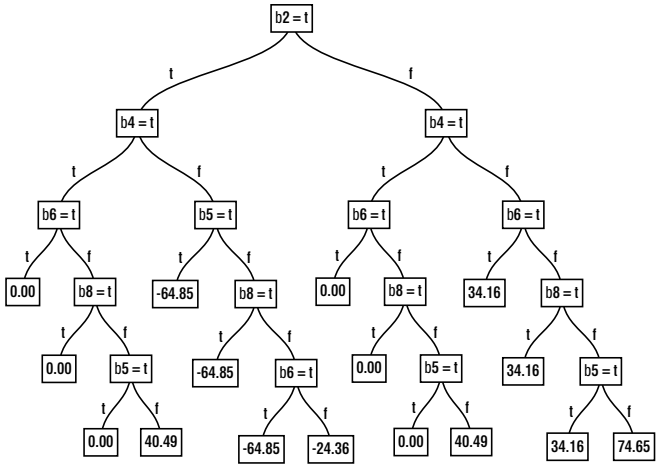

Figure 3. Regression Tree Using Variance Metric

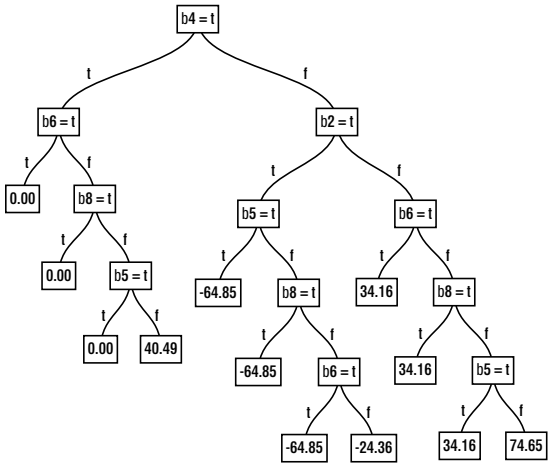

Figure 4. Regression Tree Using Unification Metric

6 Feature Construction Guided by Value Unification

We present an alternative representation and induction algorithm for finding an approximation $\hat{f}$ of target function f. The algorithm AVVU is guided primarily by value unification, and secondarily by error reduction. The representation is much more compact than that of a tree, due to the use of overlapping features.

The representation of $\hat{f}$ is a linear combination of a coefficient vector $\mathbf{W}$ and a Boolean feature vector $\mathbf{C}$ defined over $\mathbf{b}$. The overall approach is to find each term $\mathbf{w}_i\mathbf{c}_i$ one at a time using best-first search. When all the terms have been found, the process is complete. To find each term, the algorithm first finds the $\mathbf{c}_i$ feature (set cover), and then its coefficient $\mathbf{w}_i$. We describe these latter two steps first, and then return to a statement of the AVVU algorithm.

6.1 Constructing a Feature

Each feature $\mathbf{c}_i$ is a Boolean set cover, represented by a Boolean mask with the same number of components as **b**. A true value in the mask, depicted as '#' in the discussion below, matches either true '1' or false '0' in the corresponding component of **b**. A false value in the mask, depicted as '0', matches only a false '0' in the corresponding component of **b**. A feature **c** matches an instance **b** if and only if every component of **c** matches every component of **b**.

To find the next $\mathbf{c}_i$, the AVVU algorithm conducts a best-first search that is designed to construct a feature that will do the best job of unifying values. This is done indirectly by finding a **c** that eliminates overlap in the values of those training instances that are covered by $\mathbf{c}_i$ and those that are not. Overlap is measured as the number of pairs of instances of identical adjusted value that can be drawn without replacement, one instance from the covered set and one instance from the not covered set. Lower overlap is better. Indeed, the terminating condition of the best-first search is to have achieved 0 overlap.

Stated another way, we would like to unify the values of the training instances. By identifying a particular $\mathbf{c}_i$, we give ourselves the ability to adjust the values of just those instances covered by $\mathbf{c}_i$, without touching those that do not match. Associating a coefficient w_i with the feature $\mathbf{c}_i$ specifies the adjustment. We can identify a $\mathbf{c}_i$ that will best facilitate the unification process, and then determine the adjustment needed to maximize the unification of values.

Here is a trace of the AVVU algorithm for a small approximation task in which there are $d = 8$ Boolean variables, giving $2^d = 256$ possible points **b**. The target function consists of the two indicated features, making four possible f values. The 4000 training instances drawn at random contain many duplicates, and it is likely that they cover all 256 possibilities. One can see that candidate features that are more like a target feature have lower overlap. One way of seeing this is to note that a feature in the target f has the effect of adding some constant amount to the value of an instance it covers. This additive step makes a distinct set of values different from those when the value is not added. This is an assumption of the algorithm. By finding a most general feature that achieves no overlap, one has found a feature of the target function.

```
Target:
  -84.53 00##00##
  -53.16 ###000##

Expand   ########
  Create 0#######, overlap  348
  Create #0######, overlap  339
  Create ##0#####, overlap  365
  Create ###0####, overlap  266
  Create ####0###, overlap  279
  Create #####0##, overlap  255 <--- best
  Create ######0#, overlap  390
  Create #######0, overlap  397

Expand   #####0##
  Create 0####0##, overlap  136
  Create #0###0##, overlap  139
```

```
  Create ##0##0##, overlap  168
  Create ###0#0##, overlap   78
  Create ####00##, overlap   73 <--- best
  Create #####00#, overlap  182
  Create #####0#0, overlap  188

Expand   ####00##
  Create 0###00##, overlap   43
  Create #0##00##, overlap   59
  Create ##0#00##, overlap   93
  Create ###000##, overlap    0 <--- solution
  Create ####000#, overlap   92
  Create ####00#0, overlap   90

Have feature, assign weight -53.16
  -53.16 ###000##

Expand   ########
  Create 0#######, overlap  348
  Create #0######, overlap  339
  Create ##0#####, overlap  370
  Create ###0####, overlap  368
  Create ####0###, overlap  354
  Create #####0##, overlap  330 <--- best
  Create ######0#, overlap  394
  Create #######0, overlap  401

Expand   #####0##
  Create 0####0##, overlap  136 <--- best
  Create #0###0##, overlap  139
  Create ##0##0##, overlap  173
  Create ###0#0##, overlap  180
  Create ####00##, overlap  148
  Create #####00#, overlap  186
  Create #####0#0, overlap  192

Expand   0####0##
  Create 00###0##, overlap   44
  Create 0#0##0##, overlap   80
  Create 0##0#0##, overlap   81
  Create 0###00##, overlap   43 <--- best
  Create 0####00#, overlap   87
  Create 0####0#0, overlap  103

Expand   0###00##
  Create 00##00##, overlap    0 <--- solution
  Create 0#0#00##, overlap   46
```

```
  Create 0##000##, overlap   44
  Create 0###000#, overlap   48
  Create 0###00#0, overlap   50

Have feature, assign weight -84.53
  -84.53 00##00##

Approximation:
  -84.53 00##00##
  -53.16 ###000##
```

6.2 Finding the Feature's Coefficient

Suppose that the AVVU algorithm has found a feature $\mathbf{c}_i$ as described above. How can a weight be determined for it? Only a finite set of possible weights need be considered - those that can unify values of the instances covered by the $\mathbf{c}_i$ with those not covered. Any other weights would not accomplish any value unification, which would defeat the entire enterprise. The possible weights can be enumerated by considering each $f_j - f_i$, where the f_j are taken from the instances not covered, and the f_i are taken from the instances covered. One can consider the possible differences in all possible pairs of values from the two value sets. For each candidate weight, evaluate its worth by computing the size of the value set of all the training instances if the feature with that weight were appended to the approximation. The weight that produces the smallest sized value set (greatest unification) is considered best, and is adopted.

Having implemented the above, we then noticed that in practice the weight can be determined much more efficiently by taking the negative of the value that occurs most frequently in the set of instances covered by the newly constructed feature. This has the effect of unifying the largest number of values with 0, which tends to reduce overlap. We are still studying why this shortcut is so reliable.

Value unification is employed to group sets of instances in order to eliminate error. Because it is a means to an end, it can be confusing to draw a strong distinction. A natural question is how value unification manages to eliminate the error. This is because some of the training instances evaluate to 0 due to no target feature matching such an instance. When building a cover, it would be counterproductive to match such an instance, because it already evaluates correctly. Hence, there is a group of 0-error instances from the outset. Adjusting covered instances to have adjusted value 0 reduces overlap.

6.3 The AVVU Algorithm

It is now possible to state the algorithm succinctly, assuming the existence of the *find_term()* procedure, which encapsulates the two steps described above. The AVVU algorithm is:

```
Initialize f̂ to have no terms, the 0 function.
while ( size of value_set of N training points > 1 )
   { Call find_term to obtain new term.
     Append new term to f̂.
     Compute adjusted value for each training point.
```

```
    }
  return f̂
```

The algorithm finds features in general to specific order. This is important, because one would like to unify as many values as possible. If AVVU were to proceed to more specific features, it would need more of them, and therefore execute more slowly.

It is important to have an adequate sample of instances, even with a small number of features. One must sample the different regions of the instance space, even though there may be few features in the target function. We have found so far that when the algorithm does not work, increasing the sample size is the remedy.

The algorithm can be implemented efficiently. By keeping the list of instances sorted by adjusted value, one can split the list in one sweep over the instances, or join two sorted lists in one sweep. A value set can also be obtained in a single sweep. Overlap can be computed in a single sweep. The instances need to be reevaluated and sorted once at the beginning of each search for a new feature. Sorting is $O(n \log n)$, and the other operations just mentioned are $O(n)$ in the number of instances. For the step of finding the weight for a new feature, the cost of the all-differences method is very loosely $O(n^2)$ but it is much less because one is really working with value sets, which in effect discards instances with non-unique error values. These value sets shrink as the unification process progresses. However, by using the shortcut method, the cost of finding a feature is $O(n)$, because the most frequent value can be identified in one sweep.

Table 2. Target Function

```
Target:
  -93.95 #0##0####0#0#00#0###
  -41.33 #0##0###00##0####0##
  -20.82 0###00#######0###0##
  -98.97 ####0########0000###
  -45.67 #######00##0#####0#0
  -57.36 ########0##0###00###
   61.02 ##0##0####0######0##
   25.09 #######0#####0###00#
  -53.16 #0#####0########0###
  -30.68 ##########0#########
```

Table 2 shows a target function of 10 features based on 20 Boolean variables. Using the shortcut method for finding a feature's weight, and using a sample of 10,000 training instances, the AVVU algorithm found an exact solution in 29 cpu seconds. The features are shown in the table in the reverse order in which the algorithm found them.

6.4 Discussion

The AVVU algorithm produces a compact representation. For example, for the target function in Table 2, a regression tree based on the unification metric contains 879 leaves (versus 10 features), and requires 15 seconds to construct (versus 29 seconds).

The algorithm relies on the values associated with the target features being unique. No testing

has yet been done to see whether the algorithm may be able to handle non-uniqueness. The pattern of a feature is conjunctive, so there is little reason to expect that different features contributing identical amounts would be separable in this sense. We shall investigate this soon.

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.

An earlier version of the algorithm was based on a Boolean encoding of discrete variables. The algorithm did not always work for those cases. We suspect that the lack of independence of such Boolean variables is involved, but more work is needed.

We are hopeful that the algorithm will be able to tolerate noise by relaxing equality test for unification, but this has not yet been tried.

Finally, we have no proof that this algorithm will always find a good approximation. With an unrepresentative set of training points, the algorithm can become confused. More testing is needed.

We are not familiar with any other algorithms that are guided by value unification. For function approximation over a Boolean domain, one could resort to a variety of error-guided methods. For example, one could use error back-propagation with an artificial neural network (Rumelhart & McClelland, 1986). A sigmoid threshold unit can represent a Boolean set cover, but whether a network could find the needed set covers as hidden units, or find some other set of hidden units that suffice, is not known. We did train a neural network on the same 10,000 training instances for the problem Table 2, using 20 input units, 10 hidden units, and 1 output unit. Counting a network output as correct if it is within 5% of the target output, letting backprop run for 320 epochs produces a network that is 'correct' for 9501 of the 10000 examples. More epochs help little; after 20,000 epochs, correctness is only somewhat better. 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.

7 Summary

Our work presents four 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, one can benefit from trying to group instances that have similar values for similar reasons. Third, one can build smaller regression trees by using the unification metric. 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 AVVU 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 Yanfeng Lu, who provided helpful comments.

References

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

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.

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.